Avoiding Error LNK2001 Unresolved External Using DEFINE_GUID

PSS ID Number: Q130869

The information in this article applies to:

- Microsoft Visual C++ 32-bit Edition, versions 2.0, 2.1, and 4.0

SUMMARY

A GUID must be initialized exactly once. For this reason, there are two different versions of the DEFINE_GUID macro. One version just declares an external reference to the symbol name. The other version actually initializes the symbol name to the value of the GUID. If you receive an LNK2001 error for the symbol name of the GUID, the GUID was not initialized.

You can make sure your GUID gets initialized in one of two ways:

- If you are using precompiled header files, include the INITGUID.H header file before defining the GUID in the implementation file where it should be initialized. (AppWizard-generated MFC projects use precompiled headers by default.)

- If you are not using precompiled headers, define INITGUID before including OBJBASE.H. (OBJBASE.H is included by OLE2.H.)

MORE INFORMATION

Here is the definition of DEFINE_GUID as it appears in OBJBASE.H:

```
#ifndef INITGUID
#define DEFINE_GUID(name, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8) \
    EXTERN_C const GUID CDECL FAR name
#else

#define DEFINE_GUID(name, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8) \
   EXTERN_C const GUID CDECL name \
      = { l, w1, w2, { b1, b2,  b3,  b4,  b5,  b6,  b7,  b8 } }
#endif // INITGUID
```

Note that if the symbol INITGUID is not defined, DEFINE_GUID simply defines an external reference to the name.

In INITGUID.H, you find (among other things):

```
#undef DEFINE_GUID

// Other code . . .

#define DEFINE_GUID(name, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8) \
   EXTERN_C const GUID CDECL __based(__segname("_CODE")) name \
```

```
                = { l, w1, w2, { b1, b2,  b3,  b4,  b5,  b6,  b7,  b8 } }
```

By including INITGUID.H after OBJBASE.H, DEFINE_GUID is modified to actually initialize the GUID.

NOTE: It is important to make sure that this is done exactly once for each DLL or EXE. If you try to initialize the GUID in two different implementation files and then link them together, you get this error:

```
   LNK2005 <symbol> already defined.
```

Additional reference words: kbinf 2.00 2.10 4.00
KBCategory: kbole kberrmsg
KBSubcategory: VCGenIss

BUG: ASSERT in OLECLI1.CPP When Copying Embedding to Clipboard

PSS ID Number: Q152072

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SYMPTOMS

When you attempt to copy an embedding in an in-place active object to the clipboard, the result is an ASSERT in OLECLI1.CPP. Specifically, the ASSERT occurs in the COleClientItem::XOleClientSite::GetMoniker() function on the following line of code:

```
   VERIFY(pThis->m_lpObject->SetMoniker(OLEWHICHMK_OBJREL,
          *ppMoniker)==S_OK);
```

CAUSE

When you copy an embedding to the clipboard, a number of data formats are placed on the clipboard, including link source information. The link source information contains a moniker used to locate the document in which the embedding resides.

When the container assigns a moniker to an embedded object, it will call the object's IOleObject::SetMoniker() function. The object will then call the container's IOleClientSite::GetMoniker() function to construct a composite moniker based on the container's moniker. This is done because the container may have changed its moniker while the object was not running.

When the embedded object calls IOleClientSite::GetMoniker() to get the container's moniker, COleClientItem::XOleClientSite::GetMoniker() ASSERTs on the aforementioned line of code because pThis->m_bMoniker had not been set to TRUE when the moniker was assigned during a previous call to COleClientItem::XOleClientSite::GetMoniker().

The problem is located in the following section of the COleClientItem::XOleClientSite::GetMoniker() function located in OLECLI1.CPP:

```
   // notify the object of the assignment
   if (dwAssign != OLEGETMONIKER_TEMPFORUSER &&
       *ppMoniker != NULL && !pThis->m_bMoniker)
   {
      VERIFY(pThis->m_lpObject->SetMoniker(
             OLEWHICHMK_OBJREL, *ppMoniker) == S_OK);
      pThis->m_bMoniker = TRUE;
      ASSERT_VALID(pThis->m_pDocument);
```

```
      pThis->m_pDocument->SetModifiedFlag();
   }
```

The pThis->m_lpObject->SetMoniker() call results in the object calling COleClientItem::XOleClientSite::GetMoniker() again. Because pThis->m_bMoniker is being set after the pThis->m_lpObject->SetMoniker() call, the subsequent call to pThis->m_lpObject->SetMoniker() returns with an error code of E_FAIL, triggering the ASSERT.

RESOLUTION

To work around this problem, you must override the default IOleClientSite interface implementation in COleClientItem. This can be done by adding an interface map to the class in your project that is derived from COleClientItem and setting pThis->m_bMoniker to TRUE before calling pThis->m_lpObject->SetMoniker() from your custom GetMoniker() function.

The code below illustrates how to override the COleClientItem-derived object using the OCLIENT MFC sample application.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

Sample Code

```
   // Code to be added to CRectItem class definition in RECTITEM.H
   class CRectItem : public COleClientItem
   {
      // ...original declarations in CRectItem class go here...

      // Interface Map
      BEGIN_INTERFACE_PART(OleClientSite2, IOleClientSite)
         STDMETHOD(SaveObject)();
         STDMETHOD(GetMoniker)(DWORD, DWORD, LPMONIKER*);
         STDMETHOD(GetContainer)(LPOLECONTAINER*);
         STDMETHOD(ShowObject)();
         STDMETHOD(OnShowWindow)(BOOL);
         STDMETHOD(RequestNewObjectLayout)();
      END_INTERFACE_PART(OleClientSite2)

      DECLARE_INTERFACE_MAP()
   };

   // XOleClientSite2 functions to be added to RECTITEM.CPP
   BEGIN_INTERFACE_MAP(CRectItem, COleClientItem)
```

```
   INTERFACE_PART(CRectItem, IID_IOleClientSite, OleClientSite2)
END_INTERFACE_MAP()

STDMETHODIMP_(ULONG) CRectItem::XOleClientSite2::AddRef()
{
   METHOD_PROLOGUE_EX_(CRectItem, OleClientSite2)
   return pThis->ExternalAddRef();
}

STDMETHODIMP_(ULONG) CRectItem::XOleClientSite2::Release()
{
   METHOD_PROLOGUE_EX_(CRectItem, OleClientSite2)
   return pThis->ExternalRelease();
}

STDMETHODIMP CRectItem::XOleClientSite2::QueryInterface(
   REFIID iid, LPVOID* ppvObj)
{
   METHOD_PROLOGUE_EX_(CRectItem, OleClientSite2)
   return pThis->ExternalQueryInterface(&iid, ppvObj);
}

STDMETHODIMP CRectItem::XOleClientSite2::SaveObject()
{
   METHOD_PROLOGUE_EX(CRectItem, OleClientSite2)
   ASSERT_VALID(pThis);
   return pThis->m_xOleClientSite.SaveObject();
}

STDMETHODIMP CRectItem::XOleClientSite2::GetMoniker(
    DWORD dwAssign, DWORD dwWhichMoniker, LPMONIKER* ppMoniker)
{
   METHOD_PROLOGUE_EX(CRectItem, OleClientSite2)
   ASSERT_VALID(pThis);

   USES_CONVERSION;  // note, must #include afxpriv.h

   COleDocument* pDoc = pThis->GetDocument();
   ASSERT_VALID(pDoc);
   ASSERT(ppMoniker != NULL);
   *ppMoniker = NULL;

   switch (dwWhichMoniker)
   {
   case OLEWHICHMK_CONTAINER:
      // return the current moniker for the document
      *ppMoniker = pDoc->GetMoniker((OLEGETMONIKER)dwAssign);
      break;

   case OLEWHICHMK_OBJREL:
      {
         if (!pDoc->IsKindOf(RUNTIME_CLASS(COleLinkingDoc)))
         break;

         // don't return relative moniker if no document moniker
```

```
         LPMONIKER lpMoniker = pDoc-
>GetMoniker((OLEGETMONIKER)dwAssign);
         if (lpMoniker == NULL)
            break;
         lpMoniker->Release();

         // relative monikers have to handle assignment correctly
         switch (dwAssign)
         {
            case OLEGETMONIKER_ONLYIFTHERE:
               if (!pThis->m_bMoniker)
                  break;  // no moniker assigned, don't return one
                // fall through...

            case OLEGETMONIKER_TEMPFORUSER:
            case OLEGETMONIKER_FORCEASSIGN:
               {
                  // create item moniker from item name
                  TCHAR szItemName[OLE_MAXITEMNAME];
                  pThis->GetItemName(szItemName);
                  CreateItemMoniker(OLESTDDELIMOLE, T2COLE(szItemName),
                     ppMoniker);

                  // notify the object of the assignment
                  if (dwAssign != OLEGETMONIKER_TEMPFORUSER &&
                      *ppMoniker != NULL && !pThis->m_bMoniker)
                  {
                     pThis->m_bMoniker = TRUE;
                     VERIFY(pThis->m_lpObject->SetMoniker(
                         OLEWHICHMK_OBJREL, *ppMoniker) == S_OK);
                     ASSERT_VALID(pThis->m_pDocument);
                     pThis->m_pDocument->SetModifiedFlag();
                  }
               }
               break;

            case OLEGETMONIKER_UNASSIGN:
               pThis->m_bMoniker = FALSE;
               break;
         }
      }
      break;

   case OLEWHICHMK_OBJFULL:
      {
         // get each sub-moniker: item & document
         LPMONIKER lpMoniker1, lpMoniker2;
         GetMoniker(dwAssign, OLEWHICHMK_CONTAINER, &lpMoniker1);
         GetMoniker(dwAssign, OLEWHICHMK_OBJREL, &lpMoniker2);

         // create composite moniker
         if (lpMoniker1 != NULL && lpMoniker2 != NULL)
             ::CreateGenericComposite(lpMoniker1, lpMoniker2,
ppMoniker);
```

```
            // release sub-monikers
            RELEASE(lpMoniker1);
            RELEASE(lpMoniker2);
         }
         break;
      }
      return *ppMoniker != NULL ? S_OK : E_FAIL;
   }

   STDMETHODIMP CRectItem::XOleClientSite2::GetContainer(LPOLECONTAINER*
      ppContainer)
   {
   #ifdef _DEBUG
      METHOD_PROLOGUE_EX(CRectItem, OleClientSite2)
   #else
      METHOD_PROLOGUE_EX_(CRectItem, OleClientSite2)
   #endif
      ASSERT_VALID(pThis);
      return pThis->m_xOleClientSite.GetContainer(ppContainer);
   }

   STDMETHODIMP CRectItem::XOleClientSite2::ShowObject()
   {
      METHOD_PROLOGUE_EX(CRectItem, OleClientSite2)
      ASSERT_VALID(pThis);
      return pThis->m_xOleClientSite.ShowObject();
   }

   STDMETHODIMP CRectItem::XOleClientSite2::OnShowWindow(BOOL fShow)
   {
      METHOD_PROLOGUE_EX(CRectItem, OleClientSite2)
      ASSERT_VALID(pThis);
      return pThis->m_xOleClientSite.OnShowWindow(fShow);
   }

   STDMETHODIMP CRectItem::XOleClientSite2::RequestNewObjectLayout()
   {
      return E_NOTIMPL;
   }
```

Additional reference words: 1.50 1.51 1.52 2.00 2.10 2.20 4.00 4.10
KBCategory: kbole kbbuglist
KBSubcategory: MfcOLE

BUG: Blob Persistent Data Incorrect for Ported OLE Control

PSS ID Number: Q142211

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0 and 4.1

SYMPTOMS

An OLE control using PX_Blob(), created using the Control Developer's Kit (CDK) that shipped with Visual C++ 2.2 or earlier and saved in a Visual Basic 4.0 form, may give undefined behavior if the control is ported to Visual C++ 4.0 or 4.1 and the Visual Basic form containing the control is reopened.

CAUSE

OLE Controls created using the CDK that shipped with Visual C++ 2.2 or earlier did not implement the IPersistPropertyBag interface. Controls created with Visual C++ 4.0 or 4.1 do implement IPersistPropertyBag.

The undefined behavior is due to Visual Basic's implementation of IPropertyBag. Visual Basic is currently adding a 20-byte header to a blob property's persistent data, and expecting that header to be in any blob persistent data previously saved. In the case of controls created previously to Visual C++ 4.0 or 4.1, IPersistPropertyBag, which corresponds to the container's IPropertyBag was not implemented. Therefore, the container used a different means to serialize the blob data which did not attach a 20 byte header. The undefined behavior may behave a number of ways including the blob data being reinitialized, blob data being initialized incorrectly, or the control not being created altogether.

WORKAROUND

To read in the blob data correctly, you must force the container to use an interface other than IPropertyBag to serialize the data. In this case, a 20-byte header will not be expected and the blob data should be read in properly. The technique demonstrated in the "Sample Code" section of this article unexposes the control's IPersistPropertyBag interface thereby forcing the container to use an interface other than IPropertyBag to read in the blob data.

STATUS

Microsoft has confirmed this to be a bug in Microsoft Visual Basic 4.0. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

==========

Sample Code to Demonstrate Workaround

The following is a step by step process to unexpose the IPersistPropertyBag interface in an OLE control generated with Visual C++ 4.0 or 4.1. This is most easily done by implementing a new interface map, which does not expose the IPersistPropertyBag interface.

1. In the declaration of your COleControl derived class (in the header file), add a DECLARE_INTERFACE_MAP() macro if there isn't one already there.

   ```
   class CBlobTestCtrl : public COleControl
   {
   ...
       DECLARE_INTERFACE_MAP()
   };
   ```

2. In the .cpp file for the control, implement a new interface map specifying all but IID_IPersistPropertyBag.

   ```
   BEGIN_INTERFACE_MAP(CBlobTestCtrl, CWnd)  // use CWnd not COleControl
       INTERFACE_PART(CBlobTestCtrl, IID_IOleObject, OleObject)
       INTERFACE_PART(CBlobTestCtrl, IID_IConnectionPointContainer,
         ConnPtContainer)
       INTERFACE_PART(CBlobTestCtrl, IID_IOleControl, OleControl)
       INTERFACE_PART(CBlobTestCtrl, IID_IPersist, PersistStorage)
       INTERFACE_PART(CBlobTestCtrl, IID_IPersistMemory, PersistMemory)
       INTERFACE_PART(CBlobTestCtrl, IID_IPersistStreamInit,
         PersistStreamInit)
       INTERFACE_PART(CBlobTestCtrl, IID_IOleInPlaceObject,
         OleInPlaceObject)
       INTERFACE_PART(CBlobTestCtrl, IID_IOleInPlaceActiveObject,
         OleInPlaceActiveObject)
       INTERFACE_PART(CBlobTestCtrl, IID_IDispatch, Dispatch)
       INTERFACE_PART(CBlobTestCtrl, IID_IOleCache, OleCache)
       INTERFACE_PART(CBlobTestCtrl, IID_IViewObject, ViewObject)
       INTERFACE_PART(CBlobTestCtrl, IID_IViewObject2, ViewObject)
       INTERFACE_PART(CBlobTestCtrl, IID_IDataObject, DataObject)
       INTERFACE_PART(CBlobTestCtrl, IID_ISpecifyPropertyPages,
         SpecifyPropertyPages)
       INTERFACE_PART(CBlobTestCtrl, IID_IPerPropertyBrowsing,
         PerPropertyBrowsing)
       INTERFACE_PART(CBlobTestCtrl, IID_IProvideClassInfo,
         ProvideClassInfo)
       INTERFACE_PART(CBlobTestCtrl, IID_IProvideClassInfo2,
         ProvideClassInfo)
       INTERFACE_PART(CBlobTestCtrl, IID_IPersistStorage, PersistStorage)
   END_INTERFACE_MAP()
   ```

REFERENCES
==========

MFC Technical Note #38 about "MFC/OLE IUnknown implementation"
Inside Ole Second Edition by Kraig Brockschmidt

Additional reference words: 4.00 4.10
KBCategory: kbprg kbole kbbuglist
KBSubcategory: CDKIss MfcOLE

BUG: C4003 When Building Wrapper For Word.Basic 7.0 Object

PSS ID Number: Q148223

The information in this article applies to:

- The ClassWizard included with:
  Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SYMPTOMS

Attempting to build the ClassWizard-generated wrapper for the Word.Basic object by using the type library included with the Microsoft Word 95 Developer Kit (Wb70en32.tlb) results in this error:

```
warning C4003: not enough actual parameters for macro 'ExitWindows'
```

CAUSE

This problem is actually caused by a macro defined in the Windows SDK header file Winuser.h. Winuser.h redefines ExitWindows() to map to the Win32 SDK API ExitWindowsEx(). This poses a problem for any C++ classes that need to implement a member function named ExitWindows(). The Word.Basic object exposes an automation method called ExitWindows(). When ClassWizard reads the type information contained in the Word type library (Wb70en32.tlb), it generates a class containing a member function called ExitWindows(). Attempting to compile this class generates the C4003 error.

RESOLUTION

Here are two ways to work around this problem. If your application doesn't make use of the ExitWindows() SDK function, you can undefine the macro by including the following line of code before the definition of the class that contains the ExitWindows() member function.

```
#undef ExitWindows
```

However, a more straightforward workaround is to rename the ExitWindows() member function for the problem class. For example, rename WordBasic::ExitWindows() to WordBasic::_ExitWindows().

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 2.00 2.10 2.20 3.00 3.10 3.20 4.00
KBCategory: kbole kbbuglist

KBSubcategory: WizardIss

BUG: ClassWizard Incorrectly Reads LPDISPATCH Params from .TLB

PSS ID Number: Q131044

The information in this article applies to:

- The ClassWizard included with:
  - Microsoft Visual C++ for Windows, versions 1.5, 1.51, and 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0 and 2.1

SYMPTOMS

ClassWizard generates improper COleDispatchDriver derived member functions for methods that contain one or more LPDISPATCH* parameters. Specifically, ClassWizard generates member functions using a LPDISPATCH parameter instead of LPDISPATCH*. In other words, the asterisk (*) is missing.

RESOLUTION

To correct this problem, you must edit the COleDispatchDriver derived member directly.

For example, assume there is a type library (.TLB file) for an object that supports a method that returns a void and accepts an LPDISPATCH* as its only parameter. When ClassWizard reads the .TLB file, it generates a new class from COleDispatchDriver. The method for the member function that returns a void and accepts an LPDISPATCH* parameter ends up with this incorrect definition:

```
void ITestObject::TestMethod(LPDISPATCH lpDispPtr)
{
   static BYTE BASED_CODE parms[] =
      VTS_DISPATCH;
   InvokeHelper(0x1, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
      lpDispPtr);
}
```

To correct this problem, change the parameter type to LPDISPATCH*, and modify the parms[] array so that it contains a corresponding VTS_PDISPATCH entry instead of VTS_DISPATCH, as illustrated here:

```
void ITestObject::TestMethod(LPDISPATCH* lpDispPtr)
{
   static BYTE BASED_CODE parms[] =
      VTS_PDISPATCH;
   InvokeHelper(0x1, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
      lpDispPtr);
}
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.50 1.51 1.52 2.00 2.10 noupdate
KBCategory: kbole kbbuglist
KBSubcategory: WizardIss

BUG: Control Container Support Only Works in Primary Thread

PSS ID Number: Q152075

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SYMPTOMS

When using the ActiveX control container support built into MFC, controls should only be created and used in the primary thread. The primary thread is the thread created automatically for you during startup of the MFC program. Starting other threads and trying to use MFC's built-in control container support will cause unpredictable and erratic behavior of the primary thread and any other threads that attempt to display ActiveX controls. This problem will occur whether you use a UI thread or a worker thread.

CAUSE

This situation appears to occur because the afxOccManager is stored as a process local variable.

RESOLUTION

There is no reliable workaround for this issue at this time. If your program uses ActiveX controls, make sure they are displayed in the primary thread only.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

This situation can be reproduced by creating an Appwizard generated app that supports ActiveX control containment. Create a dialog in the first thread that displays an ActiveX control. Add a class to your project derived from CWinThread that will create a secondary thread. Also, add another dialog to the project that contains an ActiveX control that will be created as a modeless dialog by the secondary thread when the secondary thread starts up. This can be done in the CWinThread's InitInstance method. Build the app. When the app is run, if you show the dialog that contains the ActiveX control in the first thread, the second thread will no longer

be able to create ActiveX controls. Conversely, if the second thread shows its dialog first, the primary thread will not be able to create ActiveX controls. If you attempt to create ocx controls in more than one thread, you will also get an access violation in OLE32.dll when the program terminates.

Additional reference words: 4.00 4.10 ocx
KBCategory: kbole kbbuglist kbprg
KBSubcategory: MfcOLE

BUG: DAO SDK's CdbRecordset::Requery() May Fail

PSS ID Number: Q152318

The information in this article applies to:

- The DAO SDK included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0, 4.1

SYMPTOMS
========

Attempting to invoke the DAO SDK's CdbRecordset::Requery() method of the CdbRecordset class when the recordset has been opened with dbOpenDynaset and dbDenyWrite, may result in an error message similar to the following:

```
Unhandled exception in <YourApp>.exe (DAO3032.DLL):
0xC0000005: Access Violation
```

CAUSE
=====

Within the implementation of CdbRecordset::Requery found in DBDAO.CPP (line 1645), the casting of a COleVariant method is performed incorrectly. The cast is performed as:

```
Var.pdispVal    = (LPDISPATCH)pq();
```

The correct implementation should be:

```
Var.pdispVal    = (LPDISPATCH)pq->GetInterface();
```

RESOLUTION
==========

Derive a new class from CdbRecordset and override the Requery method with the following:

```
VOID CdbRecordset::Requery( CdbQueryDef *pq )   // = NULL
{
    DAORecordset*   prs = (DAORecordset *)GetInterface();
    COleVariant     Var;

    //Manually load the Query Def as a dispatch
    if (!pq)
    {
        Var.vt          = VT_ERROR;
        Var.scode       = DISP_E_PARAMNOTFOUND;
    }
    else
    {
        Var.vt          = VT_DISPATCH;
        Var.pdispVal    = (LPDISPATCH)pq->GetInterface();
    }
```

```
        DAOMFC_CALL(prs->Requery(Var));
    }
```

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 4.10
KBCategory: kbprg kbole kbbuglist
KBSubCategory: kbnocat

BUG: Default OLE Container w/ Splitters Faults on View Closure

PSS ID Number: Q148139

The information in this article applies to:

- The AppWizard included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS
========

An AppWizard application generated with OLE Container and Splitter Window support will generate an Access Violation when a splitter pane containing an in-place active item is closed. The Access Violation occurs when you try to call m_pView->AssertValid() from the COleClientItem::AssertValid() function.

CAUSE
=====

The AppWizard fails to add an OnDestroy() member function to the view class to deactivate the in-place active object when the view containing the in-place active object is destroyed. When the view containing the active object is destroyed, the remaining view is resized. Because the object in the destroyed view was not properly deactivated, the subsequent call to GetInPlaceActiveObject() in the view's OnSize() function erroneously detects an in-place active COleClientItem object.

RESOLUTION
==========

To work around the problem, add the following handler for the WM_DESTROY message to your view class:

```
void CYourView::OnDestroy()
{
   CView::OnDestroy();

   COleClientItem* pActiveItem =
      GetDocument()->GetInPlaceActiveItem(this);
   if (pActiveItem != NULL && pActiveItem->GetActiveView() == this)
   {
      pActiveItem->Deactivate();
      ASSERT(GetDocument()->GetInPlaceActiveItem(this) == NULL);
   }
}
```

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes

available.

MORE INFORMATION
================

Steps to Reproduce Problem
----

1. Create an SDI or MDI application with AppWizard. Select OLE Container support in Step 3, and select the "Use split window" check box in the Windows Styles tab of the Advanced Options dialog in Step 4.

2. Build a DEBUG version of the application, and run it.

3. Click Insert New Object on the Edit menu, and insert an object that supports in-place activation. For example, use Microsoft Word or Microsoft Excel. The object should appear in the container application and be activated in-place.

4. Split the container's view window. You should see two objects, one of which is in-place active and the other of which is a copy of the in-place active object.

5. Drag the splitter bar so that the view containing the in-place active object is removed and destroyed. As a result, an unhandled exception Access Violation occurs in COleClientItem::AssertValid().

Additional reference words: 4.00 GP fault GPF
KBCategory: kbole kbbuglist
KBSubcategory: WizardIss

BUG: Masked Edit Tab Order Incorrect If Visible Property FALSE

PSS ID Number: Q152603

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1, 4.2

SYMPTOMS

Multiple Masked Edit OLE Controls that are not visible will not retain their original tab order after they are made visible.

If you insert more than one Microsoft Masked Edit Ole Control into a dialog resource wrapped by a MFC Class (CDialog, CFormView, CRecordView, or CDaoRecordView), set the Visible Properties of the Controls to FALSE in the Resource Editor at design time, and then call the ShowWindow on the controls, the focus is set to the wrong control.

RESOLUTION

If possible, leave the Visible property of the Microsoft Masked Edit OLE Control set to TRUE.

If you call SetFocus on the first masked edit control, the focus is set properly to that control, but if there are any other non-static controls on the form, they will be inserted before the second masked edit control in the tab order at run time.

SetWindowPos() can be used to set the z-order of a control. The tab order in a dialog is synonymous with the Z-order, meaning that a control that is first in the tab order will have a higher Z-order than a control that follows it in the-Z order. A control with the WS_CLIPSIBLINGS window style set that is overlapped by another control will not be painted in the area that is overlapped. OLE controls have the WS_CLIPSIBLINGS window style set by default.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

Use the following steps to reproduce a dialog-based MFC application. These steps also apply if you choose either an SDI or MDI application with a CFormView, CRecordView, or CDaoRecordView:

1. Create an AppWizard dialog-based application with OLE Controls support.

2. Using the Component Gallery, insert a Microsoft Masked Edit OLE Control.

3. Place two masked edit controls on the dialog resource. Verify the tab order: the first control you placed on the dialog should be first in the tab order.

4. Set the Visible properties of both edit controls to FALSE.

5. Using the Class Wizard, create member variables of type "control" for both of the masked edit controls. For example:

   ```
   CMsmask  m_Mask1;
   CMsmask  m_Mask2;
   ```

   where CMsmask is the class that the Component Gallery created to wrap the Microsoft Masked Edit OLE Control.

6. For the dialog's OnInitDialog method, run the following code:

   ```
   m_Mask1.ShowWindow( SW_SHOW );
   m_Mask2.ShowWindow( SW_SHOW );
   ```

7. Build and run the program. The m_Mask2 object will be first in the tab order -- before m_Mask2.

Additional reference words: 4.00 4.10 4.20
KBCategory: kbusage kbole kbbuglist
KBSubcategory: MfcOLE AppStudioIss

BUG: Multiple ActiveX Control Containers Under Win32s Problems

PSS ID Number: Q152792

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SYMPTOMS

It is possible to use MFC versions 4.0 and 4.1 to create ActiveX control containers that will run under Win32s version 1.3. However if, after the container has been started under Win32s, another MFC ActiveX control container is started or a second instance of the same application is started, the second ActiveX control container can't display ActiveX controls if the first application has already displayed one of its ActiveX controls.

If the first application doesn't create its ActiveX controls first, then the second application can create ActiveX controls, but then the first application can't create ActiveX controls at all. Also, if you are running a debug build of the application, you will see an assertion in Appinit.cpp at line 110 when the second application starts up.

CAUSE

This is caused by improper initialization of the MFC and C-Run-Time DLLs under Win32s.

RESOLUTION

The assertion in Appinit.cpp at line 110 can be safely ignored. Note that assertions are only included in debug builds of MFC applications. There currently is no work around to the problem of a second MFC app being able to use MFC control container support under Win32s.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 4.10
KBCategory: kbprg kbbuglist kbole
KBSubcategory: MfcOLE VCx86

BUG: OLE Controls Limited to 20 Property Pages in Visual C++

PSS ID Number: Q153354

The information in this article applies to:

- The Resource Editor included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1, 4.2

SYMPTOMS

When an ActiveX (OLE) Control that has more than 20 Property Pages is placed on a dialog in the Visual C++ resource editor, you may encounter an error in the development environment when you switch between its property pages. In particular, if you select the tab for a property page numbered greater than 20, you will see a message box indicating: "This program has performed an illegal operation and will be shut down." Selecting a tab for a page numbered less than 20 may also cause an error to occur.

CAUSE

This behavior is caused by a bug in the Visual C++ development environment, which allows a maximum of property pages for a given ActiveX Control.

RESOLUTION

Currently, there are two workarounds for this problem:

- Limit the number of property pages for an OLE Control to 20.

  -or-

- Add a button to the property page that when clicked will bring up a Modal dialog box to handle additional properties.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

REFERENCES

OLE Controls: Adding Another Custom Property Page
Programming with MFC: Encyclopedia

Additional reference words: 4.00 4.10 4.20 vcbuglist400
KBCategory: kbole kbtool kbbuglist

KBSubcategory: AppStudioIss CDKIss

BUG: Pop-Up Menu Items Disabled for an OLE Control

PSS ID Number: Q141199

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, version 4.0 and 4.1

SYMPTOMS
========

The menu items are disabled even though menu handlers were added with Class Wizard when you create an OLE Control and then use the Pop-up Menu component in the Component Gallery to add a context menu to the COleControl derived class. The problem occurs after you modify the context menu, and then add handlers for the menu items, build the Control, place it in Tstcon32.exe, and bring up the context menu.

CAUSE
=====

Within the OnContextMenu function added by the Pop-up Menu Component, the code makes the top-most parent window the parent of the context menu. The menu sends the WM_COMMAND message to its parent first, which can then route the command appropriately through the application. This method works for windows in an MFC application that are part of the command-routing scheme. However, because OLE Controls are not part of the routing mechanism, the WM_COMMAND message never gets back to them and therefore the context menu doesn't see the handlers, so the menu items are disabled.

RESOLUTION
==========

The Pop-up Menu Component creates the following code within the CWnd derived class. If the pop-up menu is being added to a COleControl-derived class, you need to delete the following lines of code:

```
   while (pWndPopupOwner->GetStyle() & WS_CHILD)
      pWndPopupOwner = pWndPopupOwner->GetParent();
```

Code Created by Pop-up Menu Component

```
void CSPLITCtrl::OnContextMenu(CWnd*, CPoint point)
{
   // CG: This function was added by the Pop-up Menu component

   CMenu menu;
   VERIFY(menu.LoadMenu(CG_IDR_POPUP_SPLITCTRL));

   CMenu* pPopup = menu.GetSubMenu(0);
   ASSERT(pPopup != NULL);

   CWnd* pWndPopupOwner = this;
```

```
   while (pWndPopupOwner->GetStyle() & WS_CHILD)
      pWndPopupOwner = pWndPopupOwner->GetParent();

   pPopup->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON, point.x,
                          point.y, pWndPopupOwner);
}
```

By removing the two lines of code, you are leaving the COleControl-derived class as the parent of the menu. As a result, the control receives the WM_COMMAND messages sent by the pop-up menu, so the normal ON_COMMAND functionality added by Class Wizard workd as designed.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 4.10
KBCategory: kbprg kbole kbbuglist
KBSubcategory: CodeGen MfcOLE

BUG: SetWindowText(NULL) Doesn't Clear .OCX Edit Control

PSS ID Number: Q146617

The information in this article applies to:

- The Microsoft Foundation Classes included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0 and 4.1

SYMPTOMS

It is possible to use AppWizard to create an OLE Custom Control (.ocx file) that subclasses a standard Windows control. If you choose to subclass an edit control, the control will not exhibit the same behavior as a standard edit control.

Specifically, in the case where the control is not initialized with data, the user types some text into the control, and then SetWindowText(NULL) is called to clear the text, the text in the control will not be cleared. If this same sequence of events were to occur in a standard edit control, the contents of the edit control would be cleared.

CAUSE

COleControl handles the WM_SETTEXT message in its OnSetText handler. The first few lines of this function are:

```
LRESULT COleControl::OnSetText(WPARAM wParam, LPARAM lParam)
{
   ASSERT(lParam == 0 || AfxIsValidString((LPCTSTR)lParam));

   // Is the property changing?
   if ((lParam == 0 && m_strText.IsEmpty()) ||
      (lParam != 0 && m_strText == (LPCTSTR)lParam))
      return 0;
           .
         .
         .
}
```

When a user types text into the control, the m_strText member is not updated. Therefore, when SetWindowText(NULL) is called on the control, the first expression is evaluated as TRUE, even though the edit control is not truly empty. This expression is interpreted as property not changed, and the function exits.

RESOLUTION

To work around this problem, override OnSetText() in the COleControl derived class, and copy the code from COleControl::OnSetText() into the overridden funtion, with the following change:

```
LRESULT CMyOleControl::OnSetText(WPARAM wParam, LPARAM lParam)
{
   ASSERT(lParam == 0 || AfxIsValidString((LPCTSTR)lParam));

   // Is the property changing?
   if ((lParam == 0 && InternalGetText().IsEmpty()) ||
      (lParam != 0 && m_strText == (LPCTSTR)lParam))
      return 0;
                  .
         .
         .
}
```

Calling InternalGetText() will give access to the text property, and then calling IsEmpty() on the return from it will correctly check to see if the edit control has been updated.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 4.10
KBCategory: kbprg kbole kbbuglist
KBSubcategory: MfcOLE

BUG: Win32 SDK Version 3.51 Bug List - OLE

PSS ID Number: Q136433

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) for Windows NT

SUMMARY
=======

This article lists the bugs in the OLE libraries released with Windows NT version 3.51 that were known at the time the product was released.

MORE INFORMATION
================

 - IMoniker::BindToObject always returns MK_E_INVALIDEEXTENSION if an error occurs during open. It should only associate the file extension if the file is not a .doc file.

 - CreateFileMoniker expands letters to UNC names. If in doing so, the path becomes greater than MAX_PATH, CreateFileMoniker returns E_OUTOFMEMORY.

 - Cannot call CoCreateInstance for first time from the DllEntry. If you need the DLL to create an object, you can either do this by having the .exe call a creation function in your DLL or post a message to a window owned by your DLL.

 - IOleCache::SetData returns OLE_E_BLANK for invalid lIndex, FormatEtc, aspect, and target device. It should return DV_E_LINDEX, DV_E_FORMATETC, DV_E_DVASPECT, and DV_E_TARGETDEVICE, respectively.

 - Calling freeze on a frozen icon aspect with an invalid lIndex returns VIEW_S_ALREADYFROZEN rather than DV_E_INDEXFreeze of Icon aspect returns.

 - IStream::Write of 0 bytes returns E_INVALIDARG from a 16-bit OLE application running in the WOW layer. Zero is a valid parameter to pass to Write.

 - "Alt -" accelerator does not work with MDI windows.

 - IOleObject::IsUpToDate always returns S_FALSE after link track move.

 - The proxy for IEnumVARIANT::Next allows a NULL pceltFetched when celt is greater than 1.

 - If you place a dataobject which only has a TYMED_FILE format using OleSetClipboardData, GetClipboardData returns NULL.

 - The file version of IStream::CopyTo uses Windows NT signed 64-bit math to compute values it needs. Some of the math could overflow 63 bits if

the sizes and/or seek pointers of the streams are 63 bits or more.

- OleCreateFromData does not update the OBJREL moniker of the newly created embedded object.

- OleCreateLinkToFile succeeds for non-existent files. Your code should check for the existance of the file before calling this function.

- OleUIAddVerbMenu is not setting up the single menu item correctly.

- OleUIConvert and OleUIChangeIcon crash on invalid hMetaPict.

- CoFreeUnusedLibraries fails to unload DLLs that are no longer in use.

- The AddControl option, when calling OleUIInsertObject, does not work as expected.

Additional reference words:
KBCategory: kbole kbbuglist
KBSubcategory: LeTwoMisc

Calling Hidden Default Method of an OLE Automation Collection

PSS ID Number: Q152071

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SUMMARY

A technical article found on the Microsoft Developer Network CD-ROM titled "Implementing OLE Automation Collections," describes the functionality necessary to create an OLE automation collection. This includes creating a method called Item that returns the dispatch interface of the indicated item in the collection.

All OLE automation collection objects must implement the Item method to iterate through the objects of the collection. The implementation of the method might have been created hidden in the .ODL file for some applications. You cannot find these hidden methods by calling IDispatch::GetIdsOfNames(). However, in many situations, this hidden method is the default method of the collection, in which case you can invoke it with an dispatch ID of DISPID_VALUE.

Visual Basic 3.0 depends on this Item method being the default method of the collection object.

MORE INFORMATION

The following code sample illustrates using COleDispatchDriver::InvokeHelper() to call the Item method from the Resources collection class provided by Microsoft Project 4.1. The Item method is hidden and cannot be found in the type library of the object:

Sample Code

```
   void IterateCollection(void)
   {
      Application App; // wrapper class for the 'Application' object
      Project Proj;    // wrapper class for the 'Project' object
      Resources Res;   // wrapper class for the 'Resources' object
      CString lpszFileName ("c:\\temp\\project2.mpp");

      App.CreateDispatch("MSProject.Application");

      VARIANT vFileName;
      VariantInit(&vFileName);
      V_VT(&vFileName) = VT_BSTR;
      V_BSTR(&vFileName) = lpszFileName.AllocSysString();
      App.FileOpen(vFileName);
```

```
      Proj.AttachDispatch(App.GetActiveProject());
      Res.AttachDispatch(Proj.GetResources());

      VARIANT vCount;
      VariantInit(&vCount);
      V_VT(&vCount) = VT_I4;
      vCount = Res.Count();

      // the index of the collection starts with 1, not 0
      for (int i = 1 ; i <= vCount.lVal ; i++)
      {
         Resource r; // wrapper class for the 'Resource' object

         VARIANT vResult, vIndex, vStr;
         VariantInit(&vResult);
         VariantInit(&vIndex);
         V_VT(&vIndex) = VT_I2;
         V_I2(&vIndex) = i;
         static BYTE parms[] = VTS_VARIANT;

         Res.InvokeHelper(DISPID_VALUE, DISPATCH_METHOD, VT_VARIANT,
   (void*)&vResult, parms, &vIndex);
         r.AttachDispatch(vResult.pdispVal);

         vStr = r.GetName();
         // ... 'vStr' now contains the Name property
      }
   }
```

Additional reference words: 2.00 2.10 2.20 4.00 4.10
KBCategory: kbprg kbole kbcode
KBSubcategory: CodeSam MfcOLE VCx86

CONNPTS.EXE: SAMPLE: Implements Connection Points in MFC Apps

PSS ID Number: Q152087

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SUMMARY

This sample demonstrates implementing connection points and connection point sinks in MFC applications. The two applications that make up this sample are the "source" application that implements a connection point, and the "sink" application that implements an interface that will be hooked up to the connection point.

You can find CONNPTS.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile CONNPTS.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get CONNPTS.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and download CONNPTS.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download CONNPTS.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

NOTE: Use the -d option when running CONNPTS.EXE to decompress the file and recreate the proper directory structure.

MORE INFORMATION

Registration of the Connection Point Interfaces

To run the sample, the connection point interfaces must be registered because this sample demonstrates using the connection point interfaces across process boundaries and, in this case, the interfaces must be marshaled. You may or may not need to install marshaling DLLs on your system. For more information on resolving this issue, please see the following article in the Microsoft Knowledge Base:

```
   ARTICLE ID: Q149231
   TITLE     : CXPRX: Marshaling Code for Connection Point Interfaces
```

Connection Points and Sinks

For a more detailed understanding of connection points, consult the documentation in the References section below.

A COM object has interfaces that can be called by clients of the object. These are called incoming interfaces. The connection point interfaces are a means for this object to call back into the clients via outgoing interfaces. Typically, the object will want to notify any of its clients of some event or change in the object's state. The object in this case is known as the source, and the client is known as the sink.

To enable communication using the connection point mechanisms, the source object will implement one or more connection points, such as an object that supports IConnectionPoint, and the sink object will implement an interface that the source object knows how to call. It is the job of the connection point to hold on to the address of the interface in the sink object so that the source can call it. The connection point may actually hold on to more than one address if multiple sinks are clients of the object. This is called multicasting. There will be one connection point in an object for each sink interface the object knows how to call.

The remaining interface is IConnectionPointContainer. IConnectionPointContainer is implemented in the source object and serves to contain and manipulate the object's connection points.

This sample closely models the functionality of event notifications in OLE Controls. The sink in this sample implements an IDispatch interface, as would a control container, that the source knows how to use. You will need to identify this interface with a unique IID so a connection point for the interface can be uniquely identified. You will be able to take advantage of the IDispatch marshaling code to marshal the dispatch interface across process boundaries.

Note that the sink does not have to implement a dispatch interface. If you choose to implement your own interface, however, and you are going to cross process boundaries, you will be responsible for marshaling the interface.

Running the Sample

Once the connection point interfaces are registered, the sample can be run. Run the "source" application first. Each of source's open documents is an event source. Run the "sink" application next. In sink, select Sink / Attach Source from the document menu. Note that each open document in sink is an event sink. You should see a list of available sources. If you do not, the connection point interfaces are not correctly registered. Select any sources that you would like to connect to. A sink may attach itself to any number of sources and a source may have any number of sinks attached. From the source document menu, select Events / Fire Event *. Any sinks that are hooked up to the source should display a message indicating that the event was received.

Miscellaneous Notes

This sample implements connection points across process boundaries. Implementing this functionality in in-process objects is done in the same manner. In-process implementation will not require that you register the connection point interfaces or the interface that is implemented by the sink.

In this sample, the sink interface and the connection point interfaces need to be registered. This registration is taken care of in the InitInstance function of the sink application.

An IID for the sink interface is made available to both the sink and source applications. Please note that the .cpp file in which this interface is defined does not use pre-compiled headers.

All code in this sample relevant to the connection point implementation is blocked by "//SAMPLECP" and "//END SAMPLECP" blocks. This coding convention should make it easy to identify what you will need to implement in your code.

REFERENCES

"Inside OLE" by Kraig Brockschmidt, Second Edition, Chapter Four - Connectable Objects, published by Microsoft Press.

Programming with MFC: Encyclopedia - Connectable Objects

Additional reference words: 4.00 4.10 Connection Point
KBCategory: kbole kbfile
KBSubcategory: MfcOLE
words: s_mfc

ControlWizard Generates Both 16-bit and 32-bit Projects

PSS ID Number: Q125242

The information in this article applies to:

- Microsoft OLE Control Development Kit (CDK), version 1.0

SUMMARY

=======

The ControlWizard included with the Microsoft OLE Control Development Kit (CDK) generates both 16-bit and 32-bit projects. In most cases, you can develop your control on either a 16-bit or 32-bit platform, and then rebuild the control on the other platform.

MORE INFORMATION

================

When ControlWizard is used to create an OLE control, two projects are generated. The full project name you specify is used as the base name for the 16-bit MAK file and other files used by the 16-bit IDE. The first six letters of the project name followed by "32" are used to create a base name for the 32-bit MAK and DEF files and other files used by the 32-bit IDE.

For example, if the full project name is TESTCTRL, these two MAK files are generated:

```
TESTCTRL.MAK -- 16-bit Project File
TESTCT32.MAK -- 32-bit Project File
```

ControlWizard also generates a 'makefile' file that NMAKE can use to generate either a 16- or 32-bit project by specifying the WIN32 flag on the command line. For example:

```
nmake WIN32=1       // 32 bit version
nmake WIN32=0       // 16 bit version
```

Within the generated source files, place platform-dependent code in conditional compilation blocks, as shown in this example:

```
LRESULT CTestCtrl::OnOcmCommand(WPARAM wParam, LPARAM lParam)
{
#ifdef _WIN32
   WORD wNotifyCode = HIWORD(wParam);
#else
   WORD wNotifyCode = HIWORD(lParam);
#endif

   // TODO: Switch on wNotifyCode here.

   return 0;
}
```

To maintain platform independence, use conditional compilation blocks
to isolate platform dependencies in the code you add to your control.
Use the #error statement to flag features that are not implemented
for one platform or the other. For example, you might code the previous
example like the following if the 32-bit implementation was not complete:

```
LRESULT CTestCtrl::OnOcmCommand(WPARAM wParam, LPARAM lParam)
{
#ifdef _WIN32
   #error OnOcmCommand parameter decoding not completed for Win32!
#else
   WORD wNotifyCode = HIWORD(lParam);
#endif

   // TODO: Switch on wNotifyCode here.

   return 0;
}
```

Note that because separate project files are used for 16- and 32-bit
targets, all project maintenance will need to be performed for both
projects. Project maintenance activities include:

- Adding or removing project files.

- Changing compiler settings.

- Changing linker settings, including input libraries.

Additional reference words: kbinf 1.00 2.00
KBCategory: kbole kbprg kbtool
KBSubcategory: CDKIss

Debugging OLE 2.0 Applications Under Win32s

PSS ID Number: Q123812

The information in this article applies to:

- Microsoft Win32s version 1.2

SUMMARY

The following are available to help you debug your OLE 2.0 applications under Win32s:

- Debug versions of the OLE DLLs, included with Win32s. (See the Win32s Programmer's Reference for more information on the debug DLLs.)

- Failure/trace messages.

- The OLE SDK for Win32s, version 1.2, included on the Microsoft Developer Network (MSDN) CD.

For information on a utility that will convert OLE error codes into error message, please see the following article in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q122957
TITLE     : SAMPLE: DECODE32: OLE Error Code Decoder Tool
```

MORE INFORMATION

The debug version of OLE can send diagnostic information to the debug terminal. To enable this feature, include the following lines in the SYSTEM.INI:

```
[Win32sDbg]
ole20str=xxxxx
ole20str16=yyy
```

Use ole20str for 32-bit OLE and ole20str16 for 16-bit OLE. Set them to a combination of the following letters (case sensitive):

```
f - Failure message, kind of asserts.
v - Verbose. General purpose messages.
1 - Trace special translation activity for 32/16 interoperability.
i - Trace initialization of OLE.
t - Trace termination and cleanup of OLE.
```

The following tools are contained in the OLE SDK for Win32s, version 1.2:

- DFVIEW - Show the content of storage files.

- LRPCSPY - Monitor LRPC messages sent by 16-bit OLE applications (does

not require Win32s).

- RPCSPY32 - Monitor LRPC messages from both 16-bit and 32-bit OLE applications.

- DOBJVIEW and DOBJVW32 - View objects placed on the clipboard as well as objects transferred by drag and drop.

- IROTVIEW and IROTVW32 - Display the contents of the OLE running object table (ROT).

- OLE2VIEW and OLE2VW32 - Identify objects, interfaces, inproc and local servers, registration database entries, and so on.

Additional reference words: 1.20
KBCategory: kbole kbprg
KBSubcategory: W32s

DOCERR: DoDragDrop() Parameter Documented Incorrectly

PSS ID Number: Q131100

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft Visual C++ for Windows, versions 1.5, 1.51, and 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0 and 2.1

The documentation for COleClientItem::DoDragDrop(), COleServerItem::DoDragDrop(), and COleDataSource::DoDragDrop() give the following incorrect information for the lpRectStartDrag parameter:

lpRectStartDrag:

   Pointer to the rectangle that defines where the drag actually starts. It does not start until the mouse cursor leaves the rectangle. If NULL, a default rectangle is used so that the drag starts when the mouse cursor moves one pixel.

The documentation for the lpRectStartDrag parameter for DoDragDrop() should be:

lpRectStartDrag:

   Pointer to the rectangle that defines where the drag actually starts. It does not start until the mouse cursor leaves the rectangle or until a specific number of milliseconds has passed. If NULL, a default rectangle is used so that the drag starts when the mouse cursor moves one pixel. The number of milliseconds is specified by a registry key setting. The value can be found in the Windows NT system registry under the key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\NT\CurrentVersion\IniFileMapping\win.ini\Windows\DragDelay, or in the WIN.INI file under the [Windows] section if running under Windows version 3.x. If the key is not present, a default value of 200 milliseconds is used.

Additional reference words: 1.50 1.51 1.52 2.00 2.10
KBCategory: kbole kbdocerr
KBSubcategory: MfcOLE

DOCERR: GetClientRect() May Assert in OLE Controls

PSS ID Number: Q138664

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft OLE Control Developer's Kit (CDK) versions 1.0, 1.1, 1.2
  - Microsoft Visual C++, 32-bit Edition, version 4.0 and 4.1

SUMMARY

In the Ole Control Tutorial, Chapter 4. the section labeled Setting the CircleOffset Property Step 2, which describes how to implement the InBounds() function, incorrectly calls GetClientRect() and should instead read as follows:

2. Add the function implementation at the end of your CIRCCTL.CPP file:

```
BOOL CCirc2Ctrl::InBounds(short nOffset)
{
   int diameter;
   int length;
   int cx;
   int cy;

  // Note that GetControlSize() is called here instead of
  // GetClientRect()

   GetControlSize(&cx, &cy);

   if (cx > cy)
   {
      length = cx;
      diameter = cy;
   }
   else
   {
      length = cy;
      diameter = cx;
   }
   if (nOffset < 0)
      nOffset = (short) -nOffset;
   return (diameter / 2 + nOffset) <= (length / 2);
}
```

MORE INFORMATION

In some control containers (such as Visual Basic and Microsoft Access) in design mode, the control has no window, and therefore no hwnd, so calling GetClientRect(rc) will assert.

There are actually two solutions to this problem. The first solution is documented in the "Summary" section of this article. The second solution is to replace the call to GetClientRect() with a call to GetRectInContainer(), which will obtain the coordinates of the control's rectangle relative to the container. The size of the control can then be calculated from this rectangle.

Additional reference words: kbinf 1.00 1.10 1.20 4.00 4.10
KBCategory: kbole kbcode kbdocerr
KBSubcategory: CDKIss

DOCERR: How to Use the PX_Blob Function

PSS ID Number: Q137333

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft OLE Control Developer's Kit (CDK), versions 1.0, 1.1, 1.2
  - Microsoft Visual C++, 32-bit Edition, version 4.0, 4.1

SUMMARY
=======

The PX_Blob function can be used to load or save an OLE control's property that stores Binary Large Object (BLOB) data. The CDK documentation incorrectly lists the prototype for the PX_Blob function as:

```
BOOL PX_Blob( CPropExchange* pPX, LPCTSTR pszPropName,
   HGLOBAL*& hBlob, HGLOBAL hBlobDefault = NULL );
```

The third parameter is incorrectly shown as being of type HGLOBAL*&. The correct function prototype is:

```
BOOL PX_Blob( CPropExchange* pPX, LPCTSTR pszPropName,
   HGLOBAL& hBlob, HGLOBAL hBlobDefault = NULL );
```

The documentation states that PX_Blob will cause the property's value to be read from or written to the variable referenced by hBlob, as appropriate. However, the documentation is unclear about what hBlob should be referencing. In order to use PX_Blob, it is necessary that the first four bytes of information that hBlob references be a ULONG containing the number of bytes of data that make up the properties value following the ULONG.

MORE INFORMATION
================

The following steps show how to use PX_Blob to serialize BLOB data:

1. Declare a structure that contains a ULONG and the property data in the header file for the control. Because the first four bytes of a BLOB need to be a ULONG, creating a structure that contains a ULONG and the property data can simplify using PX_Blob.

2. Declare an HGLOBAL member in the control's class declaration. The HGLOBAL will be used to reference the BLOB containing the property data.

3. Initialize the HGLOBAL member to NULL in the control's constructor.

4. In the control's DoPropExchange method, check to see if the control's properties are being saved or loaded. If properties are being saved, do the following:

   a. GlobalAlloc the required number of bytes for the data that represents the property value and add four additional bytes for the ULONG that contains the size of the data. The structure declared in Step 1 can be used when doing this.

   b. GlobalLock the handle returned from the GlobalAlloc call to get a pointer of your structure type to that memory.

   c. Fill in the first part of your structure with the sizeof (your property data).

   d. Fill in the second part of your structure with the actual data that represents the control's property value.

   e. Call PX_Blob, unlock and free the global memory.

5. If the control's properties are being loaded, do the following:

   a. Call PX_Blob to get the handle to memory containing the property data.

   b. Lock the handle to get a pointer to the actual property data.

   c. Set the value of the property, and then unlock and free the global memory.

   d. If the preceding call to PX_Blob in didn't return a handle to memory containing a BLOB containing the property value, initialize the property to a default value.

The essential point in the steps is to make sure the first four bytes of information referenced by the hBlob passed to PX_Blob is a ULONG representing the number of bytes of data to follow.

Sample Code

The following sample code illustrates using the steps listed previouse in the context of the MFC CIRC3 OLE control sample. The sample code uses PX_Blob to serialize the Offset property of the CIRC3 OLE control.

```
////////////////////////////////////////////////////////////////////
// CIRC3CTL.H

//  STEP 1:
//  Declare a structure to make working with a BLOB easier.
typedef struct tagOFFSET
{
  ULONG size;
  short offset;
} OFFSET;

class CCirc3Ctrl : public COleControl
{
```

```
public:
  //  STEP 2:
  //  Declare an HGLOBAL member in the control's class declaration.
  HGLOBAL hOffset;
  ...
};

////////////////////////////////////////////////////////////////////////
// CIRC3CTL.CPP

CCirc3Ctrl::CCirc3Ctrl()
{
  InitializeIIDs(&IID_DCirc3, &IID_DCirc3Events);

  //  STEP 3:
  //  Initialize the HGLOBAL member to NULL.
  hOffset = NULL;

  //  TODO: Initialize your control's instance data here.
}

void CCirc3Ctrl::DoPropExchange(CPropExchange* pPX)
{
  ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
  COleControl::DoPropExchange(pPX);

  if (pPX->GetVersion() == (DWORD)MAKELONG(_wVerMinor, _wVerMajor))
  {
    PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);

    //****** NEW PX_Blob RELATED CODE STARTS HERE ******

    if(!pPX->IsLoading())
    {
      //  The control's properties are being saved.

      //  STEP 4.a:
      //  GlobalAlloc the size of your structure. This step could take
      //  place somewhere else and GMEM_FIXED isn't required.
      hOffset = GlobalAlloc(GMEM_FIXED, sizeof(OFFSET));

      if(hOffset != NULL)
      {
        //  STEP 4.b:
        //  GlobalLock the handle returned from the GlobalAlloc call.
        OFFSET * p_mem = (OFFSET*)GlobalLock(hOffset);

        if(p_mem != NULL)
        {
          //  STEP 4.c:
          //  Fill in the first part of your structure with the
          //  sizeof(your property data).
          p_mem->size = (long)sizeof(short); // The offset property is
                                             // a short.
          //  STEP 4.d:
```

```
      //  Fill in the second part of your structure with the actual
      //  data that represents the control's property value.
      p_mem->offset = m_circleOffset;

      //  STEP 4.e:
      //  Call PX_Blob, unlock and free your global memory.
      PX_Blob(pPX, _T("CircleOffset"), hOffset);

      GlobalUnlock(hOffset);
    }
    else
    {
      // The GlobalLock call failed. Pass in a NULL HGLOBAL for the
      // third parameter to PX_Blob. This will cause it to write a
      // value of zero for the BLOB data.
      HGLOBAL hTmp = NULL;

      PX_Blob(pPX, _T("CircleOffset"), hTmp);
    }

    GlobalFree(hOffset);
    hOffset = NULL;
  }
  else
    // The GlobalAlloc call failed. Pass in a NULL HGLOBAL for the
    // third parameter to PX_Blob. This will cause it to write a
    // value of zero for the BLOB data.
    PX_Blob(pPX, _T("CircleOffset"), hOffset);
}
else
{
  //  Properties are being loaded into the control.

  //  STEP 5.a:
  //  Call PX_Blob to get the handle to the memory containing
  //  the property data.
  PX_Blob(pPX, _T("CircleOffset"), hOffset);

  //  Definitely error check hOffset
  if(hOffset != NULL)
  {
    // STEP 5.b:
    // Lock the memory to get a pointer to the actual property
    // data.
    OFFSET * p_mem = (OFFSET *)GlobalLock(hOffset);

    if(p_mem != NULL)
    {
      //  Step 5.c:
      //  Set the value of the property, unlock and free the global
      //  memory.
      m_circleOffset = p_mem->offset;
      GlobalUnlock(hOffset);
    }
    else
```

```
          m_circleOffset = 0;

        GlobalFree(hOffset);
        hOffset = NULL;
      }
      else
        //  STEP 5.d:
        //  If the preceeding call to PX_Blob didn't give a handle
        //  to memory containing a BLOB containing the property value,
        //  initialize the property value to a default value.
        m_circleOffset = 0;
    }

    //  Comment out the original PX_Short call for the original
    //  non-BLOB version of the Offset property used by the CIRC3
    //  sample.
    // PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);

    //****** NEW PX_Blob RELATED CODE ENDS HERE ******

    PX_Long(pPX, _T("FlashColor"), (long &)m_flashColor,
        RGB(0xFF, 0x00, 0x00));
    PX_String(pPX, _T("Note"), m_note, _T(""));
  }
  else if (pPX->IsLoading())
  {
    m_circleShape = TRUE;
    m_circleOffset = 0;
    m_flashColor = RGB(0xFF, 0x00, 0x00);
    m_note = _T("");
  }
}
```

Additional reference words: kbinf 1.50 2.50 1.51 2.51 1.52 2.52 2.00 3.00 2.10 3.10 2.20 3.20 1.00 4.00 4.10
KBCategory: kbole kbcode kbdocerr
KBSubcategory: CDKIss

DOCERR: Implementing Custom Font Properties Documentation

PSS ID Number: Q127956

The information in this article applies to:

- Microsoft OLE Control Developer's Kit (CDK), versions 1.0 and 1.1

In Chapter 9 of the "CDK Programmer's Guide" from the OLE Control Development Kit Books Online, the following two lines of code are in error in the "Implementing a Custom Font Property" article:

```
   CFontHolder * m_fontHeading;

   -and-

   void CSampleControl::SetHeadingFont( LPFONTDISP HeadingFont )
```

The corrected lines should read:

```
   CFontHolder m_fontHeading;

   -and-

   void CSampleControl::SetHeadingFont( LPFONTDISP newValue )
```

In the first line, the member is an object, not a pointer. In the second line, "newValue" is the identifier name that ClassWizard generates, so you should use it unless the programmer changes the name in the argument list and in the generated function body.

Additional reference words: setfontheading set newvalue cfontholder 1.51 1.52 2.00 2.10 1.10 1.00
KBCategory: kbole kbdocerr
KBSubcategory: CDKIss

DOCERR: ThreadingModel Is Not a Subkey

PSS ID Number: Q150199

The information in this article applies to:

- The Development Environment (was Visual Workbench) included with: Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SUMMARY
=======

In the article "Processes and Threads" from the OLE Programmer's Reference, paragraph 12 gives the following incorrect information:

> For thread-aware DLL-based or in-process objects, you need to set the threading model in the registry. The default model when you do not specify a threading model is single-thread-per-process. To specify a model, you add the ThreadingModel subkey to the InprocServer32 key in the registration database.

ThreadingModel is not a subkey; it is a named value.

MORE INFORMATION
================

To set the threading model, perform these steps:

1. Run Regedit in Windows 95, or run Regedt32 in Windows NT.

2. Choose the HKEY_CLASSES_ROOT folder, and then choose the CLSID folder.

3. Find the GUID for the object you want to enable with apartment-model threading. Then expand the object's folder.

4. Expand the InprocServer32 key.

5. From the Windows 95 Edit menu, select New. Then select String Value, and type ThreadingModel. Select the name just typed from the Edit menu, select Modify, and type Apartment for Value data.

   -or-

   From the Windows NT Edit menu, select Add Value. Type ThreadingModel for the Value Name. Click OK, and type Apartment for the String value.

Additional reference words: 2.00 2.10 2.20 4.00 4.10 thread apartment
KBCategory: kbprg kbole kbdocerr
KBSubcategory: MfcThreadIss

DOCFIX: Incorrect Prototype & Memory Allocation for PX_Blob

PSS ID Number: Q135630

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft OLE Control Developer's Kit (CDK) versions 1.0, 1.1, 1.2

SUMMARY

The documentation for the prototype of PX_Blob in the CDK Books Online is shown incorrectly as:

```
BOOL PX_Blob( CPropExchange* pPX, LPCTSTR pszPropName,
   HGLOBAL*& hBlob, HGLOBAL hBlobDefault = NULL );
```

The actual prototype as shown in the CDK source code in Afxctl.h is:

```
BOOL PX_Blob(CPropExchange* pPX, LPCTSTR pszPropName, HGLOBAL& hBlob,
   HGLOBAL hBlobDefault = NULL);
```

MORE INFORMATION

Note that the third parameter is incorrectly listed as HGLOBAL*& hBlob in the documentation and should instead be HGLOBAL& hBlob as shown in the actual prototype.

There is also another error in the PX_Blob remarks section of the documentation. It states the following incorrect information:

> Note that PX_Blob will allocate memory, using the new operator, when loading BLOB-type properties. Therefore, the destructor of your control should call delete on any BLOB-type property pointers to free up any memory allocated to your control.

This correct documentation should be as follows:

> Note that PX_Blob will allocate memory, using the Windows GlobalAlloc API, when loading BLOB-type properties. You are responsible for freeing this memory. Therefore, the destructor of your control should call GlobalFree on any BLOB-type property handles to free up any memory allocated to your control.

The problems described in this article were fixed in Visual C++, 32-bit edition, version 4.0 and later.

Additional reference words: 1.50 1.51 1.52 2.00 2.10 2.20 4.00
KBCategory: kbole kbdocerr kbdocfix
KBSubcategory: CDKIss

DRAGD95.EXE:SAMPLE:OLE Drag/Drop in Windows 95 Common Controls

PSS ID Number: Q152092

The information in this article applies to:

- The Microsoft Foundation Classes, included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, and 4.1

SUMMARY
=======

This sample demonstrates implementing OLE drag and drop in Windows 95 Common controls. It also demonstrates how the drag image functionality can be preserved while dragging within the application that is the source of the data.

You can find DRAGD95.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile DRAGD95.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get DRAGD95.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and download DRAGD95.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download DRAGD95.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

NOTE: Use the -d option when running DRAGD95.EXE to decompress the file and recreate the proper directory structure.

MORE INFORMATION
================

Windows 95 Common controls implement their own style of drag and drop. However, this style of drag and drop does not support dragging to other applications. To enable this functionality, it is possible to implement OLE drag and drop using the Common controls. Unfortunately, when OLE drag and drop is implemented, the Common control drag image is lost.

It is possible, when implementing OLE drag and drop, to preserve the Common control drag image. However, this image will only be displayed when the pointer is over the application that is the source of the data. This is because the image is not a system-wide resource and belongs to the application that is the source of the data. This behavior is consistent with that of the shell in Windows 95. In Windows 95, you will notice that the drag image is lost over applications other than the Explorer or the Windows shell. The shell and any instances of the Explorer that are running are a single instance of the same application.

OLE drag and drop as implemented by this sample is straight forward. When a drag is started, a COleDataSource object is loaded with data and COleDataSource::DoDragDrop() is called. The data source is loaded with a CF_TEXT format and a private clipboard format that has been registered. Applications that understand CF_TEXT, such as Microsoft Word, can be a drop target for the data. You can also implement a COleDropTarget object so that you can be a drop target for your own custom clipboard format.

Common controls normally begin a drag operation in response to the LVN_BEGINDRAG message. You can also take advantage of this message to begin the drag operation. This is where the similarity with Common control drag and drop ends. Common control drag and drop uses mouse messages to control tracking the drag image and processing the drop. You will not be able to use mouse messages because, after you begin the drag operation, control is passed to OLE's DoDragDrop() function.

To control the tracking and display of the common control drag image, you will implement a COleDropSource object and pass it to the DoDragDrop() function. COleDropSource implements a GiveFeedback() function that is called to give feedback about the effect of a drop at the current mouse position as the mouse is moved over a drop target. Overriding the GiveFeedback() function and obtaining the position of the mouse gives you a chance to control tracking and display of the drag image.

REFERENCES
==========

Programming with MFC: Encyclopedia - Drag and Drop (OLE)

Additional reference words: 4.00 4.10
KBCategory: kbole kbfile
KBSubcategory: MfcOLE

FIX: Access Violation Firing an Error Event in OLE Control

PSS ID Number: Q141661

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2

SYMPTOMS
========

An access violation occurs when the error event is fired in an OLE Control. This occurs on the first firing of an error event in Release builds and only after several firings in Debug builds.

CAUSE
=====

An invalid "this" pointer is being loaded off the stack on a call to ExternalRelease() at the end of COleControl::FireEvent().

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Microsoft Visual C++, 32-bit Edition, version 4.0.

MORE INFORMATION
================

To reproduce this problem, add the stock Error event to an OLE Control using ClassWizard's OLE Events tab. The name given the member function to fire the event will be FireError. In the OnLButtonDown handler of the OLE Control, call FireError() and pass the following parameters:

```
     FireError(CTL_E_OUTOFMEMORY, " ");
```

This causes the SCODE of the error to be 0x800A0007 (CTL_E_OUTOFMEMORY), and the description string to be " ", a single space followed by a NULL byte. The rest of the parameters will be defaults.

Now, run the test container, insert your control, and click the control's client area. In Release builds, this causes an access violation on the first occurrence of the event. In Debug, it takes several occurrences before the access violation is seen.

Sample Code

Below is the OnLButtonDown handler for the Circ3 tutorial sample, modified to call the error event.

```
/* Compile options needed:
```

```
*/

void CCirc3Ctrl::OnLButtonDown(UINT nFlags, CPoint point)
{
   /*  Comment the tutorial.
        CDC* pdc;

   // Flash the color of the control if within the
           ellipse.
   if (InCircle(point))
   {
      pdc = GetDC();
      FlashColor(pdc);
      ReleaseDC(pdc);

      // Fire the ClickIn event
      FireClickIn(point.x, point.y);
   }
   else
      // Fire the ClickOut event
      FireClickOut();
        */

        // Here's the call.

   FireError(CTL_E_OUTOFMEMORY, " ");
   COleControl::OnLButtonDown(nFlags, point);
}
```

Additional reference words: 2.00 2.10 2.20 3.00 3.10 3.20
KBCategory: kbprg kbole kbtshoot kbbuglist kbfixlist
KBSubcategory: CDKIss

FIX: Assertion Failed Line 388 of Occmgr.cpp

PSS ID Number: Q143108

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

When an MFC 4.0 control container has one or more invisible-at-run-time OLE controls along with other OLE and non-OLE controls placed on a dialog template, the following assertion might occur based on the tab-order of the controls in the dialog:

```
Debug Assertion Failed!
Program: <app name>
File: occmgr.cpp
Line: 388
```

CAUSE

When the MFC framework creates an invisible-at-run-time OLE control, it is reparented to the desktop window. This makes the control's window not a sibling of the other windows present in the dialog. Now if this invisible-at-run-time control has another OLE control (visible at run time) next in tab order, then the framework places the latter OLE control at the end of the dialog's child window list. Later when the framework attempts to find the "next" sibling window of the OLE control (now at the end of the dialog's child window list), it returns NULL because there is no "next" sibling window to this OLE control.

RESOLUTION

The assertion can be safely ignored, but the tab order will probably be incorrect. To work around this assertion, place the invisible-at-run-time controls at the end of the tab order in the dialog template.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Visual C++ 4.1.

MORE INFORMATION

The MFC framework creates OLE controls placed on a dialog template resource, if any, by calling COccManager::CreateDlgControls. This function

in turn makes a call to COccManager::CreateDlgControl to create a specific OLE control. The code present in CreateDlgControl attempts to setup the tab order of each OLE control after it is created, for which it calls CWnd::SetWindowPos, passing in the handle of the window that the control will be inserted after (in the code, this window is referred to as pWndAfter).

For a clear understanding of what would generate the above described ASSERT, consider the following reproducible scenario.

Steps to Reproduce the Assertion Failure

Create a dialog-based control container application, and place the following controls with the specified tab order in its main dialog template resource:

- MS-COMM OLE Control (or any invisible-at-run-time control) : 1
- GRID OLE control (or any control that is visible at run time) : 2
- Static control (or any standard Windows control) : 3
- GRID OLE control (or any control that is visible at run time) : 4

Because the MS-COMM OLE control is reparented to the desktop, the GRID control next to it in the tab order will be created and placed at the end of this dialog's child window list. Later when the framework creates the final GRID control, it tries to find the static control that should precede this GRID control in the Z order, by finding the next sibling window of the second GRID control. But because there is no next sibling window to the second GRID control, it returns NULL, which triggers the ASSERT.

Additional reference words: ocx ole control cdk 4.00 4.10
KBCategory: kbprg kbole kbbuglist kbfixlist
KBSubcategory: MfcOLE CDKIss

FIX: CIRC2 and CIRC3 Samples Don't Handle WM_SIZE Message

PSS ID Number: Q138032

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft OLE Control Developer's Kit (CDK) versions 1.0, 1.1, 1.2

SYMPTOMS

Trying to decrease the size of the Circle control after changing the CircleOffset property may cause the circle to get drawn outside the control's rectangle.

CAUSE

The circle control does not respond to the WM_SIZE message and therefore does not adjust its CircleOffset property to compensate for the change in control size.

WORKAROUND

Handle the WM_SIZE message and reset the offset. This should only take place if the CircleShape property is set to TRUE and if the circle's position is outside the control's new size.

Sample Code to Implement Workaround

```
void CCirc2Ctrl::OnSize(UINT nType, int cx, int cy)
{
    COleControl::OnSize(nType, cx, cy);

    // If circle shape is true and circle does not fit in new size, reset
    // the offset

    if(m_circleShape && !InBounds(GetCircleOffset()))
       SetCircleOffset(0);
}
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in the version of the Microsoft Foundation Classes (MFC) that ships with Visual C++, 32-bit Edition, version 4.0.

MORE INFORMATION

Steps to Reproduce Problem
---

1. Build either the Circ2 or Circ3 sample.

2. On the Tools menu, click Register Control, and verify that the
   control registers successfully.

3. On the Tools menu, click Test Container.

4. On the Edit menu, click Insert OLE Control, and then click the control
   that was built in step 1.

5. Resize the control to a rather long rectangle.

6. On the View menu, click Properties, and change the CircleOffset property
   to 100 pixels to decentralize the circle.

7. Resize the control to force the circle outside the rectangle.

8. Notice that the circle is drawn outside the rectangle.

Additional reference words: 1.10 1.20 1.00 1.50 1.51 1.52 2.00 2.10 2.20
2.50 2.51 2.52 3.00 3.10 3.20 size tutorial
KBCategory: kbole kbprg kbcode kbbuglist kbfixlist
KBSubcategory: MfcOLE CDKIss

FIX: COleControl::GetNotSupported() Gives Bad Description

PSS ID Number: Q127923

The information in this article applies to:

- The OLE Control Developer's Kit (CDK), version 1.0

SYMPTOMS

If you define a property using the Get or Set method and call COleControl's GetNotSupported() member function from your Get<propname> method implementation, the error that is thrown uses the AFX_IDP_E_SETNOTSUPPORTED define instead of the correct AFX_IDP_E_GETNOTSUPPORTED as the description identification.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in the OLE Control Developer's Kit version 1.1.

Additional reference words: 1.00 1.51 2.00
KBCategory: kbole kbbuglist kbfixlist
KBSubcategory: CDKIss

FIX: CWnd::SubclassDlgItem Returns FALSE for OLE Controls

PSS ID Number: Q148703

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

Using CWnd::SubclassDlgItem to subclass an OLE Control returns FALSE, after failing on the check for m_pCtrlCont != NULL in line 301 of Winctrl1.cpp.

CAUSE

The MFC framework's implementation of CWnd::SubclassDlgItem checks and uses the CWnd::m_pCtrlCont member of the OLE control, instead of the control container, which is the parent of the control. The source code should use pParent->m_pCtrlCont in order to access the container.

RESOLUTION

As a workaround, use CWnd::SubclassWindow instead of CWnd::SubclassDlgItem, which is illustrated in the sample code section in this article.

STATUS

Microsoft has confirmed this to be bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Microsoft Visual C++, 32-bit Edition, version 4.1.

MORE INFORMATION

Steps to Help Reproduce Problem

1. Generate a default SDI control container application using AppWizard.

2. Insert the Grid OLE Control through the Component Gallery, which in turn generates the CGridCtrl wrapper class.

3. Place the Grid control in the Application's About dialog box.

4. Declare an embedded member variable called "m_gridctrl" in CAboutDlg of type CGridCtrl.

5. Generate an OnInitDialog() handler for the CAboutDlg class and

subclass the Grid control inside it as illustrated by this code:

```
BOOL CAboutDlg::OnInitDialog()
{
    // Call base class implementation
    CDialog::OnInitDialog();

    // Try to subclass the Grid control by calling SubclassDlgItem
    m_gridctrl.SubclassDlgItem(IDC_GRID1, this);

    return TRUE;  // set focus to first control of the dialog
}
```

NOTE: In this code, IDC_GRID1 is the ID of the Grid control.

6. When running the container application through the debugger, notice that CWnd::SubclassDlgItem returns FALSE after failing on the check for m_pCtrlCont != NULL in line 301 of Winctrl1.cpp.

Sample Code to Work Around Problem

```
BOOL CAboutDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Try to Subclass the Grid control by calling SubclassWindow
    CWnd* pWnd = GetDlgItem(IDC_GRID1);
    m_gridctrl.SubclassWindow(pWnd->m_hWnd);

    return TRUE;  // set focus to first control of the dialog
                  // Exception: OCX Property Pages should return FALSE
}
```

Additional reference words: 4.00 ocx ole control
KBCategory: kbprg kbole kbcode kbbuglist kbfixlist
KBSubcategory: MfcOLE CDKIss

FIX: DisplayAsDefault Ambient Property Not Updated for Control

PSS ID Number: Q148616

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

When you use the arrow keys to change between controls in a Visual C++ 4.0 generated container, the DISPID_AMBIENT_DISPLAYASDEFAULT ambient property of the control site is not updated, and any control that relies on this property doesn't function correctly.

For example, a subclassed button control that is programmed to update its border according to the DISPID_AMBIENT_DISPLAYASDEFAULT ambient property will exhibit this behavior. In a Visual C++ generated container, the control won't update its border.

CAUSE

In Occdlg.cpp, the following line 542 shouldn't be called:

```
bCheckDef = FALSE;
```

This line keeps CheckDefPushButton from being called and the ambient property from being updated.

RESOLUTION

The best solution is to upgrade to Visual C++ 4.1. If this is not a possibility, the MFC DLL can be rebuilt without bCheckDef = FALSE; however, Microsoft recommends against this because you need to rebuild the DLL with a different name (see tech note 33).

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Microsoft Visual C++ version 4.1.

MORE INFORMATION

Sample Code

The following code can be added to the Button Control sample shipped

with Visual C++ to demonstrate this problem.

```
////////////////////////////////////////////////////////////////////////
/
// Display the button with a thick border if appropriate

void CButtonCtrl::OnAmbientPropertyChange(DISPID dispid)
{
    TRACE("OnAmbientPropertyChange called\n");
    BOOL bDisplayAsDefault = FALSE;
    if(DISPID_AMBIENT_DISPLAYASDEFAULT != dispid)
        return;
    // Check the control site to see if you should add or remove the border
    if (!GetAmbientProperty(DISPID_AMBIENT_DISPLAYASDEFAULT, VT_BOOL,
            &bDisplayAsDefault))
        ASSERT(FALSE);
    if(bDisplayAsDefault)
        SendMessage(BM_SETSTYLE, BS_DEFPUSHBUTTON, 0);
    else
        SendMessage(BM_SETSTYLE, BS_PUSHBUTTON, 0);
    InvalidateControl();
}
```

Additional reference words: 4.00 focus
KBCategory: kbole kbprg kbbuglist kbfixlist
KBSubcategory: MfcOLE

FIX: Error in COleDispatchException Constructor

PSS ID Number: Q140590

The information in this article applies to:

- Standard and Professional Editions of Microsoft Visual Basic programming system for Windows, version 4.0

- The Microsoft Foundation Classes (MFC) included with:

  - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SYMPTOMS

In an MFC application, AfxThrowOleDispatchException is used to throw an exception from within an OLE Automation function. When Microsoft Visual Basic is used as the automation client application, it will not be able to correctly interpret the error code passed to AfxThrowOleDispatchException.

CAUSE

AfxThrowOleDispatchException constructs and throws a COleDispatchException object, which is used to handle exceptions specific to the OLE IDispatch interface. The constructor of COleDispatchException incorrectly initializes the COleDispatchException object, resulting in the Visual Basic client being unable to interpret the error code of the exception.

RESOLUTION

To implement the proper behavior for throwing an OLE dispatch exception from an MFC server, do not use AfxThrowOleDispatchException. Instead, construct your own COleDispatchException, set m_scError = 0, and throw the exception yourself as illustrated in the sample code in this article.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Visual C++ 4.1.

MORE INFORMATION

COleDispatchException has two public members (m_wCode and m_scError) which are mapped by COleDispatchException::Process() to the wCode and scode members of an EXCEPINFO structure. An EXCEPTINFO structure is used by an automation object to describe an exception that occurred during IDispatch::Invoke. When an automation object raises an exception, it should

store an error code inside either wCode or scode but not both. If an error code is passed to the constructor of the COleDispatchException object, the values of both wCode and scode in the EXCEPINFO structure will be set.

Sample Code

```
// Insert the following code instead of AfxThrowOleDispatchException
COleDispatchException* pException;
pException = new COleDispatchException(
                 _T("Some Exception"), 200, 425);
pException->m_scError = 0;

THROW(pException);
```

REFERENCES
==========

"Inside OLE" second edition by Kraig Brockschmidt, published by Microsoft Press, Chapter 14, pages 658 - 660.

Additional reference words: 1.50 1.51 1.52 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.00 2.0 2.1 2.2 1.50 1.51 1.52 4.10
KBCategory: kbole kbprg kbbuglist kbfixlist
KBSubcategory: MfcOLE

FIX: Link Paste Update Causes ASSERT in AFXWIN1.INL

PSS ID Number: Q130867

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, version 2.0
  on the following platform(s): x86

SYMPTOMS

Some containers respond to change notifications in Paste Link objects by calling GetData() on the linked object with a pointer to a valid DVTARGETDEVICE. This is usually a printer device or something other than the display context, which can be created by the server. In the case where the container is a 16-bit Windows-based application, this may cause an assertion in AFXWIN1.INL line 457.

CAUSE

In the case of a 16-bit Windows-based container application, MFCANS32.DLL attempts to translate the fields of the 16-bit DVTARGETDEVICE structure to 32-bit equivalents and pass the new structure on to the 32-bit MFC-based server application. In doing this, it uses a copy of the DVTARGETDEVICE structure containing mis-ordered fields. Specifically, the tdDeviceNameOffset and the tdDriverNameOffset fields are reversed in order. This causes any server that attempts to use the DVTARGETDEVICE to create a valid device context to fail when it uses these fields to call ::CreateDC().

In the MSVC20\MFC\SRC\OLEMISC.CPP module, the utility function _AfxOleCreateDC() does not check the return code from ::CreateDC() and returns an invalid HDC as a result. Later, that handle is attached to a CDC object. When AFXWIN1.INL's CDC::GetDeviceCaps() function is called, it validates the HDC member that was attached, and that is where the assertion occurs. Depending on the circumstances and container application being used, the symptom may be different.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Microsoft Visual C++, 32-bit Edition, version 2.1 for x86 platforms.

Additional reference words: 2.00 createdc dvtargetdevice mfcans32
KBCategory: kbole kbfixlist kbbuglist
KBSubcategory: MfcOLE

FIX: LNK1152 & LNK1141 When Linking OLE Control to Mapi32.lib

PSS ID Number: Q141485

The information in this article applies to:

- The MAPI32.LIB included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

Attempting to link an OLE Control project with Mapi32.lib results in the following warnings and errors:

```
warning LNK4022: cannot find unique match for symbol "DllCanUnloadNow"
warning LNK4002: _DllCanUnloadNow defined in C:\MSDEV\LIB\MAPI32.LIB
warning LNK4002: _DllCanUnloadNow@0 defined in
                 C:\MSDEV\MFC\lib\mfcs40d.lib
warning LNK4002: _DllCanUnloadNow@0 defined in
                 C:\MSDEV\LIB\oleaut32.lib
warning LNK4022: cannot find unique match for symbol
                 "DllGetClassObject"
warning LNK4002: _DllGetClassObject defined in C:\MSDEV\LIB\MAPI32.LIB
warning LNK4002: _DllGetClassObject@12 defined in
                 C:\MSDEV\MFC\lib\mfcs40d.lib
warning LNK4002: _DllGetClassObject@12 defined in
                 C:\MSDEV\LIB\oleaut32.lib

fatal error LNK1152: cannot resolve one or more undecorated symbols
fatal error LNK1141: failure during build of exports file
```

CAUSE

The import library for the Mapi32.dll (Mapi32.lib) incorrectly exports the DllCanUnloadNow and DllGetClassObject functions. These functions should be exported privately and not included in the import library.

RESOLUTION

To work around this problem, build the Mapi32.lib import library from scratch by using the Lib.exe utility included with Visual C++ 4.0.

Copy the sample code listed below into a text file, and save it as Mapi32.def. From the command prompt, enter the following command line to execute the Lib.exe utility and build a new Mapi32.lib import library:

```
LIB /DEF:MAPI32.DEF
```

Replace the incorrect Mapi32.lib file located in the ~\Msdev\Lib directory with this newly created Mapi32.lib file, and rebuild the OLE Control project.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Visual C++ 4.1.

MORE INFORMATION
================

For additional information on how to build import libraries without access to source files, please refer to the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q131313
   TITLE   : How to Create 32-bit Import Libraries Without .OBJs
               or Source

Sample Code

```
; Mapi32.def : Used DUMPBIN /EXPORTS to determine the functions and
; ordinal values listed in the EXPORTS section below. Note
; DllCanUnloadNow, DllGetClassObject, and WEP are NOT listed below.

LIBRARY      "MAPI32.DLL"

EXPORTS
      BMAPIAddress      @35
      BMAPIDetails      @37
      BMAPIFindNext     @34
      BMAPIGetAddress   @36
      BMAPIGetReadMail  @33
      BMAPIReadMail     @32
      BMAPIResolveName  @38
      BMAPISaveMail     @31
      BMAPISendMail     @30
      MAPIAddress       @19
      MAPIDeleteMail    @17
      MAPIDetails       @20
      MAPIFindNext      @16
      MAPIFreeBuffer    @18
      MAPILogoff        @12
      MAPILogon         @11
      MAPIReadMail      @15
      MAPIResolveName   @21
      MAPISaveMail      @14
      MAPISendDocuments @10
      MAPISendMail      @13
```

Additional reference words: LNK1152 LNK1141 LNK4022 LNK4002 4.00 Win32 sdk ocx ole control cdk 4.10
KBCategory: kbole kbbuglist kbfixlist kbcode
KBSubcategory: MfcOLE CDKIss

FIX: Rebuilding .CLW File Does Not Restore Link to ODL File

PSS ID Number: Q135049

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 2.0 and 2.1

SYMPTOMS

A project's Object Description Language (ODL) file is not updated when new automation properties and methods are added using ClassWizard. The project's ClassWizard (.clw) file has been deleted and rebuilt prior to the occurrence of this behavior. All properties and methods added before the rebuilding of the ClassWizard file are still correctly listed in the ODL file.

CAUSE

When the .clw file is rebuilt by the development environment, the ODLFile= keyword line is not added to the new file.

RESOLUTION

The ODLFile= entry must be manually added to the .clw file so that ClassWizard can add properties and methods to the ODL file. The ODLFile= entry can be added by using the development environment editor or Notepad.

Manually adding the ODLFile= entry will not automatically update the ODL file to bring it back in synch with ClassWizard. Any properties and methods that were added via ClassWizard while the ODLFile= entry was missing from the .clw file must be either manually added to the ODL file or removed and then added again by using the ClassWizard.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Visual C++ version 2.2 for Windows NT.

MORE INFORMATION

An example of the ODLFile= line can be found in the .clw file provided with the AutoClik sample. In this sample, the [General Info] section of the Autoclik.clw file includes the following line:

```
ODLFile=autoclik.odl
```

ClassWizard uses this reference to a specific ODL to determine the file to

update when new properties and methods are added. When the .clw file is rebuilt using the ClassWizard option from the development environment Resource Editor, the ODLFile= reference line is not included in the newly built .clw file. After the rebuild, ClassWizard does not issue a warning when adding new properties and methods to the new .clw file even though the changes are not reflected in the ODL file.

Additional reference words: 2.00 2.10
KBCategory: kbtool kbole kbfixlist kbbuglist
KBSubcategory: WizardIss

FIX: Scribble File Extension Conflicts with Screen Savers

PSS ID Number: Q126751

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  - Microsoft Visual C++ for Windows, versions 1.50, 1.51, and 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0 and 2.1

SYMPTOMS

Creating or activating embedded or linked OLE objects-based files saved with the MFC server sample "Scribble" fails in unexpected ways. Two common scenarios are:

- When you attempt to insert an object into an OLE container using the Create from File option, instead of creating a new Scribble Document object and inplace activating it, the Packager is invoked and a package is created. Nothing appears to happen when you try to activate the package.

- While a Scribble Document is open in the Scribble application, you copy the contents to the clipboard, and attempt to paste link from an OLE container. The link is created and the container document's image of the link displays correctly. But after the container document is saved and both the container and Scribble applications are shut down, when you reopen the saved document, the container is unable to update or activate the link to the Scribble Document.

CAUSE

The .SCR file extension used by Scribble is associated with another file type, typically screen saver applications.

RESOLUTION

Change the file extension used by Scribble Documents to something that is not already associated with a file type, for example .SCB. After changing the file extension, you will need to run Scribble once in standalone mode so it can update the system registry.

The file extension can be changed by editing the string resource IDR_SCRIBTYPE in the Scribble application. Initially the string value is:

```
\nScrib\nScrib\nScrib Files (*.scr)\n.SCR\nScribble.Document.1\n
   Scrib Document\nSCRI\nscri Files
```

To change the file extension to .SCB, edit the string value to

look like this:

```
   \nScrib\nScrib\nScrib Files (*.scb)\n.SCB\nScribble.Document.1\n
      Scrib Document\nSCRI\nscri Files
```

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Microsoft Visual C++, 32-bit edition, version 4.0.

MORE INFORMATION
================

MFC Applications can update the file associations maintained in the system registry by calling CWinApp::RegisterShellFileTypes(). However, RegisterShellFileTypes() does not replace an existing association. Thus when an association already exists for the .SCR file extension, nothing is added to the registry to indicate that Scribble should be used to create or edit these files.

This situation is most likely to occur on Windows 95, as .SCR files are associated with Screen Saver applications when Windows 95 is installed.

Additional reference words: 1.50 2.50 1.51 2.51 1.52 2.52 2.00 3.00 2.10 3.10
KBCategory: kbinterop kbole kbbuglist kbfixlist
KBSubcategory: MfcOLE

FIX: TypeLibs with Large Argument Lists May Crash ClassWizard

PSS ID Number: Q148230

The information in this article applies to:

- The ClassWizard included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

The Developer Studio crashes or abruptly terminates when attempting to add a new class from an OLE TypeLib with the ClassWizard.

CAUSE

ClassWizard allocates a 1024 byte buffer to store member function prototypes based on the information it finds in the specified OLE TypeLib. If the ClassWizard-generated prototype for a member function exceeds this limit, Developer Studio will crash or abruptly terminate.

This problem only occurs when attempting to create a new class from an OLE TypeLib, where one or more of the automation methods contains a large argument list. For example, both Microsoft Schedule+ 7.0 and Microsoft Project 4.0 expose automation methods with large argument lists. Attempting to create a new class from either the Schedule+ type library (SP7EN32.OLB) or the Microsoft Project type library (PJ4EN32.OLB) may cause Developer Studio to crash.

When the crash occurs, you will notice that a header (.H) file and an implementation (.CPP) file were added to your project's directory. These files will most likely be corrupted and should be deleted from your project directory.

RESOLUTION

Avoid using the ClassWizard to generate new classes with the previously listed type libraries. Microsoft has provided a self-extracting file, called TLBWRAP.EXE, that contains the COleDispatchDriver-derived classes that wrap the automation objects exposed by the SP7EN32.OLB and PJ4EN32.OLB type libraries.

You can find TLBWRAP.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile TLBWRAP.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)

ftp ftp.microsoft.com
Change to the Softlib/Mslfiles folder
Get TLBWRAP.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and Download TLBWRAP.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download TLBWRAP.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Visual C++ 4.1.

Additional reference words: 4.00 4.10 softlib software library
KBCategory: kbtool kbfixlist kbbuglist kbole kbfile
KBSubcategory: WizardIss MfcOLE

How To Access Binary Data Using dbDao

PSS ID Number: Q152294

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SUMMARY

When using the DAO SDK C++ classes to access binary data (such as a bitmap) you will find that the data is returned in a COleVariant. COleVariant is an MFC class that wraps the OLE VARIANT data type. Within the VARIANT, the data is stored as an OLE SAFEARRAY.

Extracting the binary data from the COleVariant requires some knowledge of VARIANTs and SAFEARRAYs. The sample code below illustrates how to work with these data types by providing a function for extracting binary data from a COleVariant and a function for storing binary data in a COleVariant.

MORE INFORMATION

Sample Code

```
   //Extensive error checking is left out to make the code more readable

   BOOL GetBinaryFromVariant(COleVariant & ovData, BYTE ** ppBuf,
                             unsigned long * pcBufLen)
   {
     BOOL fRetVal = FALSE;

   //Binary data is stored in the variant as an array of unsigned char
     if(ovData.vt == (VT_ARRAY|VT_UI1))  // (OLE SAFEARRAY)
     {
       //Retrieve size of array
       *pcBufLen = ovData.parray->rgsabound[0].cElements;

       *ppBuf = new BYTE[*pcBufLen]; //Allocate a buffer to store the data
       if(*ppBuf != NULL)
       {
         void * pArrayData;

         //Obtain safe pointer to the array
         SafeArrayAccessData(ovData.parray,&pArrayData);

         //Copy the bitmap into our buffer
         memcpy(*ppBuf, pArrayData, *pcBufLen);

         //Unlock the variant data
         SafeArrayUnaccessData(ovData.parray);
```

```
      fRetVal = TRUE;
    }
  }
  return fRetVal;
}

BOOL PutBinaryIntoVariant(COleVariant * ovData, BYTE * pBuf,
                          unsigned long cBufLen)
{
  BOOL fRetVal = FALSE;

  VARIANT var;
  VariantInit(&var);  //Initialize our variant

  //Set the type to an array of unsigned chars (OLE SAFEARRAY)
  var.vt = VT_ARRAY | VT_UI1;

  //Set up the bounds structure
  SAFEARRAYBOUND  rgsabound[1];

  rgsabound[0].cElements = cBufLen;
  rgsabound[0].lLbound = 0;

  //Create an OLE SAFEARRAY
  var.parray = SafeArrayCreate(VT_UI1,1,rgsabound);

  if(var.parray != NULL)
  {
    void * pArrayData = NULL;

    //Get a safe pointer to the array
    SafeArrayAccessData(var.parray,&pArrayData);

    //Copy bitmap to it
    memcpy(pArrayData, pBuf, cBufLen);

    //Unlock the variant data
    SafeArrayUnaccessData(var.parray);

    *ovData = var;  // Create a COleVariant based on our variant
    fRetVal = TRUE;
  }

  return fRetVal;
}


//How you might call these functions

CdbRecordset rs;

//Code for initializing DAO and opening the recordset left out...

COleVariant ovData = rs.GetField(_T("MyBinaryField"));
```

```
   BYTE * pBuf = NULL;
   unsigned long cBufLen;

   if(GetBinaryFromVariant(ovData,&pBuf,&cBufLen))
   {
     //Do something with binary data in pBuf...

     //Write back a new record containing the binary data

     COleVariant ovData2;
     if(PutBinaryIntoVariant(&ovData2,pBuf,cBufLen))
     {
       rs.AddNew();
       rs.SetField(_T("MyBinaryField"), ovData2); //Write our COleVariant
   to the table
       rs.Update();
     }
     //Clean up memory allocated by GetBinaryFromVariant
     if(pBuf)
       delete pBuf;
   }
```

Additional reference words: 4.00 4.10
KBCategory: kbole kbprg kbhowto
KBSubcategory: dbDao

How To Debug OLE Server Applications Using MSVC

PSS ID Number: Q151074

The information in this article applies to:

- Microsoft OLE libraries included with:
  - Microsoft Windows NT, version 3.51
  - Microsoft Windows 95, version 4.0

SUMMARY
========

An OLE client application involves interaction with other OLE server applications. This interaction could be with in-process or out-of-process servers. The client application may or may not have debugging information. These combinations make the debugging process of an OLE application complicated. This article presents some techniques that can be used for debugging OLE-enabled applications.

MORE INFORMATION
=================

Debugging an In-Process OLE Server Application

An OLE client application interacting with an in-process OLE server application is simply loading an OLE server DLL in the client application address space. To debug such an in-process server, standard DLL debugging techniques can be used. For more information, please see the following article in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q85221
TITLE     : Debugging a Dynamic-Link Library (DLL) in Windows
```

Debugging an Out-of-Process OLE Server Application

In an OLE client application interacting with an out-of-process OLE server application, the debugging involves crossing process spaces, which makes it much more difficult. Following are few techniques that can be used to debug out-of-process OLE server applications:

- Setup a hardcoded breakpoint in the server, and when the breakpoint hits in the server code, the Microsoft Visual C++ (MSVC) debugger is launched. Then step through server code and add breakpoints at locations of interest in server code.

  Add the following line of code in the source code, where a hardcoded breakpoint is needed.

  ```
  DebugBreak();
  ```

- Setup a breakpoint in the container/client application, then step into the code where the server code is called. The action of stepping into the code causes a new version of the MVSC debugger to be launched with the Server debug information. Then you can step through the server code and add breakpoints at locations of interest in the server code. To configure this, check in the MSVC option, the Tool Option, the debug tab, the Just-in-time debugging, and the OLE RPC debugging checkboxes that need to be checked. This technique requires source code and a debug version of the client application as well. For more information, please see the following article in the Microsoft Knowledge Base:

    ARTICLE-ID: Q122680
    TITLE   : Tutorial: Debugging OLE Applications

- Launch the server application from the client and then find the process id of the server application using PView. Launch the MSVC -p <process ID>. This launches the MSVC option that attaches itself to the running server application. However, this technique is not useful in debugging the startup code of the server application. For more information, please see the following article in the Microsoft Knowledge Base:

    ARTICLE-ID: Q120707
    TITLE   : How to Debug an Active Process in Visual C++ 2.0

- If the server application has registered the ClassFactory object with the system as single use (CoRegisterClassObject with CLS_SINGLEUSE flag), you can run the server application from the MSVC option as stand alone. To simulate the server being launched from the container/client application, you need to specify the /Embedding /Automation program arguments as applicable in the MSVC debug options and run the server application as stand alone. The /Embedding and /Automation switches do the appropriate server initialization, and register the class factory as if launched from the container/client application. Because the server's class factory is single use and not already connected, when the container/client application tries to hook up to the server, the container connects to the running server application in the debugger, and debugging is easier. For more information, please see the following article in the Microsoft Knowledge Base:

    ARTICLE-ID: Q117558
    TITLE   : Tutorial: Debugging OLE Client-Server Combinations

- If the server application registers the ClassFactory object with the system as multiple use (CoRegisterClassObject with CLS_MULTIPLEUSE flag), run the server application from the MSVC option as stand alone. Since such servers register the ClassFactory on startup, you do not need to specify /Embedding /Automation program arguments in the MSVC debug options. When the container/client application tries to connect to the server, it connects to the instance of the server application in the debugger because the server's class factory is registered multiple use, and debugging is easier.

Additional reference words: 3.50 3.51 4.00
KBCategory: kbole kbtshoot kbhowto
KBSubcategory: LeTwoOth

How to Disable the Resizing of an OLE Control

PSS ID Number: Q137538

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft OLE Control Developer's Kit (CDK) versions 1.0, 1.1, 1.2

SUMMARY
=======

To disable the resizing of an OLE Control, you need to override the COleControl::OnNcLButtonDown() and COleControl::OnSetExtent() functions.

MORE INFORMATION
================

The default implementation of COleControl::OnNcLButtonDown() initiates the tracker rectangle and changes the extent of the control based on the results.

The default implementation of COleControl::OnSetExtent() is called by the framework to change the size of the control based, at least in part, on the sizing handled by the client container.

To disable resizing, you need to disable the tracker rectangle supported by COleControl::OnNcLButtonDown() and ignore any size changes sent to COleControl::OnSetExtent().

To disable resizing in the Circle3 tutorial, override CCirc3Ctrl::OnSetExtent() and CCirc3Ctrl::OnNcLButtonDown() and implement as follows:

```
BOOL CCirc3Ctrl::OnSetExtent(LPSIZEL lpSizeL)
{
   return FALSE;
}

void CCirc3Ctrl::OnNcLButtonDown(UINT nHitTest, CPoint point)
{
   return;
}
```

Additional reference words: kbinf resize static fixed 1.50 1.51 1.10 1.20 1.00 1.52 2.00 2.10 2.20
KBCategory: kbprg kbole kbcode
KBSubcategory: CDKIss MfcOLE

How to Display an MFC Automation Document Automatically

PSS ID Number: Q140591

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY
=======

Some OLE Automation servers, like Microsoft Excel, automatically display a worksheet when it is created through OLE Automation. OLE Automation servers implemented in MFC do not display the document window by default when the document object is created through OLE Automation. This article explains how to add this behavior to a generic MFC OLE Automation server created using AppWizard.

MORE INFORMATION
================

The creation process for a document object created through OLE Automation slightly differs from the process involved when the server's File.New menu item is used. The document object is created using a class factory, which is represented by the application object's COleTemplateServer data member (m_server).

When a new document object is created through OLE Automation, COleTemplateServer::OnCreateObject() is called, which calls CDocTemplate::OpenDocumentFile with the last parameter set to FALSE to create an invisible document window. By replacing this call to CDocTemplate::OpenDocumentFile() with the last parameter set to TRUE, a visible document window will be created.

Additionally, the code in the application's InitInstance(), which checks if the server is running embedded or automated, should be removed. The main window of the application will not be displayed if this code isn't removed and the server isn't already running.

Step-by-Step Procedure

The following steps illustrate how to add this behavior to a generic MFC OLE Automation server created using AppWizard:

1. Using AppWizard, create an SDI or MDI application called MyApp with OLE Automation support.

2. Add the following class definition before the project's CWinApp-derived class's definition in the application object's header file (MyApp.h).

```
// CMyTemplateServer used to override OnCreateObject.
class CMyTemplateServer : public COleTemplateServer
{
   // Constructors / Destructors
   public:
   CMyTemplateServer(){}
   virtual ~CMyTemplateServer(){}

   // Overrides
   virtual CCmdTarget* OnCreateObject();
};
```

3. Add the following implementation of OnCreateObject() to the CWinApp-derived class's implementation file (MyApp.cpp).

```
// Identical to COleTemplateServer::OnCreateObject() except for
// the call to OpenDocumentFile() with the last parameter set to
// TRUE instead of FALSE.
CCmdTarget* CMyTemplateServer::OnCreateObject()
{
   ASSERT_VALID(this);
   ASSERT_VALID(m_pDocTemplate);

   // Save application user control status
   BOOL bUserCtrl = AfxOleGetUserCtrl();

   // Create visible doc/view/frame set
   CDocument* pDoc;
   pDoc = m_pDocTemplate->OpenDocumentFile(NULL, TRUE);

   // Restore application's user control status
   AfxOleSetUserCtrl(bUserCtrl);

   if (pDoc != NULL)
   {
      ASSERT_VALID(pDoc);
      ASSERT_KINDOF(CDocument, pDoc);

      // All new documents created by OLE start out modified
      pDoc->SetModifiedFlag();
   }
   return pDoc;
}
```

3. Change the CWinApp-derived object's m_server member from type COleTemplateServer to CMyTemplateServer. For example:

```
class CMyApp : public CWinApp
{
    public:
    CMyApp();

    // Overrides
    // ClassWizard generated virtual function overrides
    // {{AFX_VIRTUAL(CMyApp)
```

```
    public:
    virtual BOOL InitInstance();
    // }}AFX_VIRTUAL

    // Implementation
    CMyTemplateServer m_server; // <== change to derived class
    ...
    ...
};
```

4. Remove the check for running embedded or automated from the application's InitInstance() member function. For example, remove the following lines of code from your application's InitInstance() function:

   ```
   // Check to see if launched as an OLE server
   if (cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated)
   {
      // Application was run with /Embedding or /Automation.
      // Don't show the main window in this case.
      return TRUE;
   }
   ```

   NOTE: This code is specific to Visual C++ version 4.0; it varies with the earlier versions.

5. Compile the application, and run it stand alone.

6. Create an OLE Automation client that creates an instance of your document type. Upon creation of the document object, the view should automatically be displayed.

Additional reference words: kbinf 1.50 1.51 1.52 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.0 2.0 2.1 2.2 1.50 1.51 1.52
KBCategory: kbole kbprg kbhowto kbcode
KBSubcategory: MfcOLE

How to Draw Controls in an OLE Metafile

PSS ID Number: Q127192

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++ for Windows, versions 1.5, 1.51, and 1.52
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 4.0, and 4.1 on the following platform(s): x86

SUMMARY
=======

You can use controls, either directly on a CView or from a dialog template on a CFormView, in an OLE enabled MFC application. However, when the item is embedded but not active, the controls on these views will not be drawn to the Windows metafile supplied to the COleServerItem::OnDraw() function. In this case, you must "draw" the controls manually into the metafile. This article shows you how.

MORE INFORMATION
================

The simplest way to get controls to "draw" in a metafile is to use the same method the VIEWEX sample uses to draw its CInputView to a printer device context. VIEWEX's OnPrint routine draws each control as a rectangle, circle, text, and so on. Override the COleServerItem::OnDraw() function and insert your code into it. As an example, use the COleServerItem::OnDraw() function listed in this article; it shows both the VIEWEX code to insert in the OnDraw member function and the helper function PaintChildWindows() that actually does the painting of each control.

Sample Code

```
/* Compile options needed: Standard
*/

void CMyServerItem::OnDraw(CDC* pDC) // pDC is actually a metafile
{
    //BLOCK: Set up scale mode
    {
        CClientDC dcScreen(NULL);
        pDC->SetMapMode(MM_ANISOTROPIC);
        // map 1 screen logical inch to 1 printer (/output) logical inch
        pDC->SetWindowExt(dcScreen.GetDeviceCaps(LOGPIXELSX),
                dcScreen.GetDeviceCaps(LOGPIXELSX));
        pDC->SetViewportExt(pDC->GetDeviceCaps(LOGPIXELSX),
                pDC->GetDeviceCaps(LOGPIXELSX));
    }
    // we must also offset the window positions relative to the scroll
    // offset
```

```
    // We cheat here since some controls do not paint if they are
    // invisible, so we temporary make set the appropriate visible bits
    // during preview mode so the controls think they are visible even
    // though they aren't.

    HWND hWndCheatVisible = NULL;
    if (!IsWindowVisible())
    {
        // walk up to the top until we find the invisible window
        for (HWND hWnd = m_hWnd;
            hWnd != NULL; hWnd = ::GetParent(hWnd))
        {
            ASSERT(hWnd != NULL);
            DWORD dwStyle = ::GetWindowLong(hWnd, GWL_STYLE);
            if ((dwStyle & WS_VISIBLE) == 0)
            {
                ::SetWindowLong(hWnd, GWL_STYLE, dwStyle | WS_VISIBLE);
                hWndCheatVisible = hWnd;
                break;
            }
        }
        ASSERT(hWndCheatVisible != NULL);
    }

    CPen pen(PS_SOLID, 1, RGB(0,0,0));  // solid black pen
    CPen* pOldPen = pDC->SelectObject(&pen);

    ASSERT(pDC->GetWindowOrg() == CPoint(0,0));
    CRect pRect = new CRect(-50,-50,600,600);

    PaintChildWindows(m_hWnd, pDC, GetDeviceScrollPosition());
    ASSERT(pDC->GetWindowOrg() == CPoint(0,0));
    pDC->SelectObject(pOldPen);

    if (hWndCheatVisible != NULL)
        ::SetWindowLong(hWndCheatVisible, GWL_STYLE,
            ::GetWindowLong(hWndCheatVisible, GWL_STYLE) &~ WS_VISIBLE);
}

void CMyServerItem::PaintChildWindows(HWND hWndParent,
                                      CDC* pDC, CPoint ptOffset)
{
    for (HWND hWndChild = ::GetTopWindow(hWndParent);
        hWndChild != NULL;
        hWndChild = ::GetNextWindow(hWndChild, GW_HWNDNEXT))
    {
        CRect rect;
        ::GetWindowRect(hWndChild, rect);  // wnd rect in screen coords
        ScreenToClient(&rect);             // relative to this view

        HDC hdcOut = pDC->m_hDC;
#ifdef _DEBUG
        CPoint pt = pDC->GetWindowOrg();
        ASSERT(pt.x == 0 && pt.y == 0);
```

```
#endif

        DWORD dwStyle = ::GetWindowLong(hWndChild, GWL_STYLE);
        if (dwStyle & (WS_HSCROLL|WS_VSCROLL))
        {
            TRACE("Warning: printing control with scrollbars not
 supported\n");
        }
        if (dwStyle & WS_BORDER)
        {
            // the only case we special case - manually drawn border
            ::Rectangle(hdcOut, rect.left, rect.top, rect.right,
                        rect.bottom);
            rect.InflateRect(-1,-1);        // 1 logical pixel
        }

        pDC->SaveDC();
        {
            CPoint pt(ptOffset.x + rect.left, ptOffset.y + rect.top);
            pDC->LPtoDP(&pt);
            pDC->OffsetViewportOrg(pt.x, pt.y);
                // set the viewport origin so that the window origin
                //  can be changed by the control

            // draw it using a non-virtual HDC
            ::SendMessage(hWndChild, WM_PAINT, (WPARAM)hdcOut, 0L);
        }
        pDC->RestoreDC(-1);

        if (::GetTopWindow(hWndChild) != NULL)
            PaintChildWindows(hWndChild, pDC, ptOffset);
    }
}
```

Additional reference words: mfc ole inplace embed 1.50 2.00 2.10 2.50 2.51 2.52 3.00 3.10 4.00 4.10
KBCategory: kbole kbcode
KBSubcategory: MfcOLE

How to Enumerate OLE and VB Controls from an OLE Control

PSS ID Number: Q141414

The information in this article applies to:

- Standard and Professional Editions of Microsoft Visual Basic programming system for Windows, version 4.0

- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY

In order for an OLE control to communicate with another control placed on the same Visual Basic form, the OLE control needs access to at least one interface pointer of the other control. This article illustrates a technique you can use to enumerate both OLE and Visual Basic controls present on a particular form and retrieve an interface pointer to these controls. Note that the same technique could be applied to enumerate controls placed on other control containers, provided that container exposes the functionality required to implement the technique.

MORE INFORMATION

Given a pointer to IOleClientSite, it is possible to enumerate through all of the other controls on a form by making use of the following interfaces:

IOleClientSite
IOleContainer
IEnumUnknown
IUnknown
IOleObject
IOleClientSite
IOleControlSite

Note that most of these interfaces are container-side interfaces, so the technique mentioned here is container dependent. For this method to work, the container must provide support for IOleContainer, which is currently defined as a mandatory interface in the OLE control container guidelines. Both MFC version 4.0 and Visual Basic version 4.0 OLE control containers provide support for this interface. Before using this method with another control container, ensure that it provides support for IOleContainer.

The method itself is illustrated by the sample code listed in this article. You can incorporate the code into an OLE control generated using ControlWizard. Use the code to enumerate all the controls, internal Visual Basic controls as well OLE controls, by calling IOleContainer::EnumObjects, and passing the following flags as its first parameter:

```
   OLECONTF_EMBEDDINGS: is used to retrieve OLE Controls.
   OLECONTF_OTHERS    : is used to retrieve other objects such as Visual
                        Basic internal controls.
```

```
   hr = lpContainer->EnumObjects(OLECONTF_EMBEDDINGS | OLECONTF_OTHERS,
                                 &lpEnumUnk);
```

The differentiating aspect between OLE controls and other objects such as internal Visual Basic controls is that only OLE controls support the IOleObject interface. Hence, if a QueryInterface for IID_IOleObject fails for an object, then it is a different type of object. Also, if the control container provides support for Extended controls as does Visual Basic 4.0, the Extended control for a particular OLE control can also be retrieved using the method illustrated by the sample code.

Note that most of the functionality provided by the following sample code depends solely on the extent of the functionality exposed by the control container itself.

Sample Code

```
void EnumAllControlNames(LPOLECLIENTSITE lpSite)
{
   LPOLECONTAINER lpContainer;
   LPENUMUNKNOWN lpEnumUnk;

   // Note that the IOleContainer interface is currently defined as
   // mandatory. It must be implemented by control containers,
   // in the OLE Control Containers Guidelines.
   HRESULT hr = lpSite->GetContainer(&lpContainer);
   if(FAILED(hr)) {
      OutputDebugString(_T("Unable to get to the container.\n"));
      return;
   }

   // OLECONTF_EMBEDDINGS is used to retrieve OLE Controls.
   // OLECONTF_OTHERS is used to retrieve other objects such as
   // Visual Basic internal controls
   hr = lpContainer->EnumObjects(
                      OLECONTF_EMBEDDINGS | OLECONTF_OTHERS,
                      &lpEnumUnk);
   if(FAILED(hr)) {
      lpContainer->Release();
      return;
   }

   LPUNKNOWN lpUnk;
   while (lpEnumUnk->Next(1, &lpUnk, NULL) == S_OK) {
      LPOLEOBJECT lpObject = NULL;
      LPOLECONTROLSITE lpTargetSite = NULL;
      LPOLECLIENTSITE lpClientSite = NULL;
      LPDISPATCH lpDisp;

      hr = lpUnk->QueryInterface(
                    IID_IOleObject, (LPVOID*)&lpObject);
      if(SUCCEEDED(hr)) {
         // This is an OLE control.
```

```
   // Navigate to the Extended Control because Visual Basic 4.0 uses
   // Extended controls.
   hr = lpObject->GetClientSite(&lpClientSite);
   if(SUCCEEDED(hr)) {
      // You have the IOleClientSite interface
      hr = lpClientSite->QueryInterface(
                IID_IOleControlSite, (LPVOID*)&lpTargetSite);
      if(SUCCEEDED(hr)) {
         // You have the IOleControlSite interface
         // Get the IDispatch for the extended control.
         // Note that Extended controls are optional in the OLE
         // specifications for OLE Control Containers.
         hr = lpTargetSite->GetExtendedControl(&lpDisp);
      }
   }
}
else {
   // This is either an internal VB control or the
   // VB form itself.
   hr = lpUnk->QueryInterface(
                IID_IDispatch, (LPVOID*)&lpDisp);
}

if(SUCCEEDED(hr)) {
   VARIANT va;
   VariantInit(&va);
   DISPID dispid;
   DISPPARAMS dispParams = { NULL, NULL, 0, 0 };

   // Get the names of all the controls present in a VB form.
   LPWSTR lpName[1] = { (WCHAR *)L"Name" };
   hr = lpDisp->GetIDsOfNames(IID_NULL, lpName, 1,
                              LOCALE_SYSTEM_DEFAULT, &dispid);

   if(SUCCEEDED(hr)) {
      hr = lpDisp->Invoke(dispid/*0x80010000*/, IID_NULL,
                          LOCALE_SYSTEM_DEFAULT,
                          DISPATCH_PROPERTYGET |
                          DISPATCH_METHOD,
                          &dispParams, &va, NULL, NULL);
      if(SUCCEEDED(hr)) {
         CString szTmp((LPCWSTR)va.bstrVal);
         // szTmp now has the name.
         OutputDebugString(_T("And the name is ... ") + szTmp +
                           _T("\n"));
      }
   }
   lpDisp->Release();
}

// Release interface pointers.
if(lpObject)     lpObject->Release();
if(lpTargetSite) lpTargetSite->Release();
if(lpClientSite) lpClientSite->Release();
```

```
      lpUnk->Release();
   }    // End of While statement

   // Final clean up
   lpEnumUnk->Release();
   lpContainer->Release();
}
```

Additional reference words: kbinf 2.00 2.10 2.20 4.00 VB VC visualc cdk ocx
KBCategory: kbole kbprg kbhowto kbcode
KBSubcategory: MfcOLE CDKIss

How to Gain Access to an OLE Control from Its Property Page

PSS ID Number: Q143432

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++ for Windows, versions 1.0, 1.5, 1.51, 1.52
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY

When using an OLE control, you find situations where there is a need to call member functions or gain access to member variables of the control derived class from its associated property page. This can be achieved by making use of the array of IDispatch pointers (held by each property page) that represent the objects being affected due to the manipulations done through the property page. This article explains in detail how this can implemented and gives a code sample to illustrate it.

MORE INFORMATION

Property sheets, in an OLE control, allow an end user to directly manipulate the control's properties by displaying one or more property pages that display a collection of properties. These properties could belong either to one particular control or to a collection of OLE controls.

Each OLE control property page is an in-proc object with its own CLSID that implements the interface IPropertyPage. The IPropertyPage::SetObjects member function is used to provide a property page with pointers to the objects (IUnknowns) manipulated by this particular page. Please refer to the OLE Programmer's Reference, Vol. 1, for more information about the SetObjects function.

The MFC implementation for the IPropertyPage interface stores the object pointers as an array of IDispatchs representing the controls that are affected by a particular property page. This array can be accessed by using COlePropertyPage::GetObjectArray(). The property pages in MFC make use of this IDispatch array to apply the changes directly to those objects (that is, the controls) by creating a COleDispatchDriver-derived class, attaching the IDispatch to this class, and invoking the SetProperty/GetProperty of COleDispatchDriver to convey the change to the control-derived class.

An OLE Control generated using the ControlWizard creates a property page that can be used to manipulate the properties of one particular OLE control rather than manipulating a collection of controls. Hence, the control associated to a property page can be accessed by obtaining the previously mentioned IDispatch array in the COlePropertyPage and calling the static function CCmdTarget::FromIDispatch to return a pointer to the CCmdTarget object associated with any one of the IDispatchs. The sample code section of this article illustraties this method.

Note that calling CCmdTarget::FromIDispatch(), for an IDispatch pointer belonging to an OLE Control, will always return NULL in versions before MFC 4.x. For more information about this problem, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q138414
   TITLE   : PRB: FromIDispatch Returns NULL for OLE Control

This is no longer a problem in versions MFC 4.x.

Sample Code

```
// The header file of the control-derived class must be included in
// the same source file.
#include "myctrl.h"

CMyCtrl* CMyPropPage::GetControlClass()
{
  CMyCtrl *pMyCtrl;
  ULONG Ulong;

  // Get the array of IDispatchs stored in the property page
  LPDISPATCH FAR *m_lpDispatch = GetObjectArray(&Ulong);

  // Get the CCmdTarget object associated to any one of the above
  // array elements
  pMyCtrl = (CMyCtrl*) CCmdTarget::FromIDispatch(m_lpDispatch[0]);

  // Cleanup
  return pMyCtrl;
}

// If your control has a public member variable, in this case
// I am using m_direct_control, then that variable can be
// manipulated as follows.

void CMyPropPage::OnLButtonDown(UINT nFlags, CPoint point)
{
  // Modify a member variable of Control directly.
  CMyCtrl *pMyCtrl = GetControlClass();
  pMyCtrl->m_direct_control++;

  // Display the new value of the variable in a message box.
  char buf[100];
  AfxMessageBox (_itoa (pMyCtrl->m_direct_control, buf, 10));

  COlePropertyPage::OnLButtonDown(nFlags, point);
}
```

In this code, it is assumed that the array of IDispatchs returned from GetObjectArray holds the same IDispatch pointer because in a default ControlWizard-generated application, each property page manipulates a particular OLE control.

Additional reference words: kbinf 1.00 1.50 2.00 2.10 2.51 2.52 3.00 3.10 3.20 4.00 ocx visualc
KBCategory: kbole kbprg kbhowto kbcode
KBSubcategory: CDKIss MfcOLE

How to Handle Events for OLE Controls in a CWnd

PSS ID Number: Q147740

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, version 4.0

SUMMARY
=======

OLE Controls are most often used in CDialog- or CFormView-derived classes. There are occasions that call for an OLE Control to be a child of a window that is not associated with a dialog template, such as a CWnd. Adding event handlers for a control created as a child of a CWnd requires a slightly different approach than adding the same handlers for the control when placed on a dialog. This article provides step-by-step instructions for adding event handlers to the CWnd-derived class that parents the control.

MORE INFORMATION
================

Step-by-Step Example

Use the following steps to create an OLE control dynamically in a CWnd-derived window. The control used here is the Circ3 control from the Circle Tutorial, but the approach is valid for any OLE control. After creating the control, handlers are created for Circ3's ClickIn and ClickOut events.

1. Create the application that is to contain the OLE Control. This application must include support for OLE controls. Select the window that is to be the parent of the OLE control. This window could be an SDI or MDI view or any window of a CWnd-derived class.

2. Add the Circ3 control from the OLE controls Tab in the Component Gallery. At this point, ClassWizard will add a wrapper class for the Circ3 control and will add Circ3 to the toolbar in the dialog editor.

3. On the Insert menu, click Resource, and then click Dialog to insert a new dialog resource into the project. Select Circ3 from the toolbar, and add it to your dialog. Note the control ID will be something like IDC_CIRC3CTRL1.

4. With the dialog still open in the editor, press CTRL+W to bring up ClassWizard. At this point, you will see the "Select a Class" dialog. Choose "Select an existing class," and specify the CWnd-derived class. Click Yes to confirm that this is not a dialog class.

5. When you will see the control's events listed in the messages window, select ClickIn, and click ADD FUNCTION. Then select ClickOut, and click ADD FUNCTION. ClassWizard adds the appropriate EVENTSINK_MAP, ON_EVENT macros, and event handlers to the class specified in Step 4.

NOTE: Add all necessary handlers at this time. In order to add more handlers later, you must delete the dialog resource and repeat this process.

6. Add a CCirc3 member variable named m_circ to the CWnd-derived class declaration as shown below. You must include the header file of the CCirc3 wrapper class for this declaration:

```
CCirc3 m_circ;
```

7. Use ClassWizard to override the Create() virtual function for the CWnd-derived class. In the handler, call Create() to create the Circ3 child control as shown below.

NOTE: If you are creating the OLE Control in a view, you may override OnInitialUpdate() instead of Create().

```
BOOL MyCWnd::Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName,
  DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID,
  CCreateContext* pContext)
{
BOOL result = CWnd::Create(lpszClassName, lpszWindowName, dwStyle,
  rect, pParentWnd, nID, pContext);

if(result != 0)   //Create the Circ3 OLE Control
  result = m_circ.Create("Test", WS_VISIBLE, CRect(1,1,130, 120),
     this, IDC_CIRC3CTRL1);

return result;
}
```

IMPORTANT: Parameter 5 is the ID for the control. This ID must match the ID used in the ON_EVENT macro created by ClassWizard. This is the ID of the Circ3 control created in Step 3.

8. After adding the necessary event handlers, you may delete the temporary dialog resource from the project.

Optional: Another approach often used to add event handlers to CWnd-derived classes is to add controls and event handlers to a dialog class, and then cut and paste the appropriate code into the View.h and View.cpp files making modifications as necessary. Note, however, that the previous method is less prone to errors.

REFERENCES
==========

"MFC 4.0 Helps You Contain Your OLE Controls," MSDN Nov/Dec 1995.

"Handling Events from Dynamically Instantiated OLE Controls," Microsoft Visual C++ Version 4.0 Technical Information and White Papers.

OLE Control Containers: Programming OLE Controls in an OLE Control Container - Visual C++ Books Online, MFC Encyclopedia

Additional reference words: kbinf 4.00
KBCategory: kbole kbhowto kbcode
KBSubcategory: CDKIss MfcOLE

How to Handle OCM_CTLCOLOREDIT Reflected Message

PSS ID Number: Q148242

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SUMMARY
=======

This article shows you how to change the background color of an OLE control that subclasses an Edit control.

MORE INFORMATION
================

To change the background color of an OLE Control that subclasses an Edit Control, you must handle the OCM_CTLCOLOREDIT(32-bit) messages. These messages are intercepted by the "reflector window" (created for an OLE control that subclasses a Windows control) which reflects them back to the OLE control itself. In response to these reflected messages, you must set the background color (and optionally the foreground color) and return a handle to a brush initialized with the background color.

Step-by-Step Example

The sample code in this example illustrates how to handle OCM_CTLCOLOREDIT in order to change the background color of an OLE control that subclasses an Edit control

1. Generate a Control Wizard Application, and select the option that allows you to subclass an Edit control.

2. To handle an OCM_CTLCOLOREDIT reflected window message, declare the following handler function in the .h file of your control's class:

   ```
   LRESULT OnOcmCtlColor(WPARAM wParam, LPARAM lParam);
   ```

3. In the .cpp file of your control's class, add an ON_MESSAGE entry to the message map:

   ```
   ON_MESSAGE(OCM_CTLCOLOREDIT, OnOcmCtlColor)
   ```

4. Also in the .cpp file, implement the OnOcmCtlColor member function to process the reflected message:

   ```
   LRESULT CEdtclrCtrl::OnOcmCtlColor(WPARAM wParam, LPARAM lParam)
   {
     //Declare CBrush* m_pBackBrush in your control's .h file
     if (m_pBackBrush == NULL)
        m_pBackBrush = new CBrush(RGB(0,0,0));
   ```

```
    CDC* pdc = CDC::FromHandle((HDC)wParam);
    pdc->SetBkMode(TRANSPARENT);
    pdc->SetBkColor(RGB(0,0,0));
    pdc->SetTextColor(RGB(0,255,0));
    HBRUSH far* hbr = (HBRUSH far*)m_pBackBrush->GetSafeHandle();
    return ((DWORD)hbr);
  }
```

   NOTE: In your control's constructor, set m_pBackBrush = NULL, and in your control's destructor, delete m_pBackbrush.

5. Build and register your control.

6. Insert this control into the Test Container. Notice that the background color of your OLE control is changed.

REFERENCES
==========

Refer to Handling Reflected Windows Messages in OLE Controls::Subclassing a Windows Control in MFC4.0-> Programming with MFC:Encyclopedia.

Additional reference words: 4.00
KBCategory: kbprg kbole kbhowto kbcode
KBSubcategory: MfcOLE

How to Implement Per-Property Browsing for a Custom Property

PSS ID Number: Q140592

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY

This article illustrates how to implement per-property browsing for a custom property present in an OLE control. The custom property can be manipulated either in a property page that implements editing of this property or directly in the property browser provided by the container application.

When the property browser of Microsoft Visual Basic is used, for example, selecting the custom property for editing displays a three-dot button next to the property's value. Clicking the three-dot button displays the property page associated with the property -- if one is available.

MORE INFORMATION

An OLE control container typically provides some kind of user interface where the user can manipulate the properties of the control. Some containers may need to browse individual property values rather than groups of property values, which is a technique commonly referred to as per-property browsing. An OLE control can support non-default, per-property browsing by implementing IPerPropertyBrowsing. Otherwise, the container application uses the type information.

When an OLE control's property is manipulated using the property browser provided by a container application, the container queries to find out if the control supports IPerPropertyBrowsing. If an interface pointer can be successfully obtained, the container then makes a call to IPerPropertyBrowsing::MapPropertyToPage to obtain the CLSID of the property page that implements editing of this particular property. If this method returns S_OK or S_FALSE, the container creates a property frame with the property page corresponding to the returned CLSID in it.

The success return codes (S_OK and S_FALSE) from IPerPropertyBrowsing::MapPropertyToPage specify whether or not a property can be edited outside the property page identified by the CLSID. A return code of S_FALSE implies that this particular property can only be edited through a specific property page -- not outside this property page. For example, the property cannot be directly edited in the property browser of Visual Basic. If the method returns S_OK, then the property can be manipulated outside the property page. Please refer to the CDK Books Online for more information on MapPropertyToPage and its return values.

The sample code in this article can be added to an OLE control to implement a custom property that can only be edited using a specific property page -- not outside the property page. The code overrides two member functions of COleControl, namely OnMapPropertyToPage and OnGetDisplayString. The MFC framework calls OnMapPropertyToPage to obtain the CLSID of a property page that implements editing of the property identified by a dispID. The framework calls OnGetDisplayString to obtain a string representing the property's value to be displayed in a container-supplied property browser.

Steps to Add Custom Property Named MyProp to an OLE Control

For the code to work, the ClassWizard must have been used to add a custom property named MyProp to an OLE control. The following steps illustrate how to add a custom property named MyProp using ClassWizard. This will use the Get/Set Methods implementation:

1. Load the OLE control project.

2. Open ClassWizard.

3. Click the OLE Automation tab.

4. Click the Add Property button.

5. In the External name box, type MyProp.

6. Under Implementation, select Get/Set Methods.

7. From the type box, select short as the property's type.

8. Type unique names for the Get and Set methods, or accept the default names, and then click OK.

9. Click the OK button to confirm the choices, and close ClassWizard.

Sample Code

Once the MyProp property is added, the following code can be used to display that property in the property browser provided by Visual Basic:

```
BOOL CTestCtrl::OnMapPropertyToPage(DISPID dispid, LPCLSID
 lpclsid, BOOL* pbPageOptional)
{
     switch (dispid)
     {
      case dispidMyProp:
         // Return the CLSID of the property page that implements
         // editing of the MyProp property
         *lpclsid = CTestPropPage::guid;
         pbPageOptional = FALSE; // Can't be edited outside this page
         return TRUE;
     }
```

```
     return COleControl::OnMapPropertyToPage(
                         dispid, lpclsid, pbPageOptional);
}

BOOL CTestCtrl::OnGetDisplayString(DISPID dispid, CString& strValue)
{
     switch (dispid)
     {
      case dispidMyProp:
         // Return any string that should be displayed in the property
         // browser provided by VB
         strValue = _T("(My Property)");
         return TRUE;
     }

     return COleControl::OnGetDisplayString(dispid, strValue);
}
```

Additional reference words: kbinf 1.51 1.52 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.0
KBCategory: kbprg kbole kbhowto kbcode
KBSubcategory: MfcOle CDKIss

How to Implement Scaled Printing in an MFC/OLE Container

PSS ID Number: Q138266

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, and 4.1

SUMMARY

This article shows by example how to implement scaling properly.

Printing an embedded object from an AppWizard-generated OLE container application may cause the printout of the embedded object to appear too small.

The developer of an OLE container must implement the code to scale an embedded object properly from the screen device to the printer device. If only the screen device is considered, the embedded object will appear too small on a printout.

MORE INFORMATION

Sample Code

/* Compile options needed: none
*/

The MFC sample Contain, Step 2 implements a container that does not consider the difference in resolution between the screen device and the printer device. If, for instance, Scribble, Step 7 is embedded in Contain and PRINT or PRINT PREVIEW is selected, the embedded object will appear too small.

Using Contain as an example of MFC container that doesn't support scaling, the following code can be added to CContainView::OnDraw() to implement proper scaling.

```
void CContainView::OnDraw(CDC* pDC)
{
        CContainDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        // draw the OLE items from the list
        POSITION pos = pDoc->GetStartPosition();
        while (pos != NULL)
        {
```

```
                // draw the item
                CCntrItem* pItem = (CCntrItem*)pDoc->GetNextItem(pos);

                 // copy the client items rect, which will be scaled
                 CRect rect2(pItem->m_rect);
                 if(pDC->IsPrinting())
                 {
                         // get the printer's pixel resolution
                         int ixp=pDC->GetDeviceCaps(LOGPIXELSX);
                         int iyp=pDC->GetDeviceCaps(LOGPIXELSY);

                 // get the current window's resolution
                         CDC* pddc = GetDC();
                         int ixd=pddc->GetDeviceCaps(LOGPIXELSX);
                         int iyd=pddc->GetDeviceCaps(LOGPIXELSY);

                 // scale rect2 up by the ratio of the resolutions
                         int width=rect2.Width()*(ixp/ixd);
                         int height=rect2.Height()*(ixp/ixd);
                         rect2.right  = rect2.left +
                                     MulDiv(rect2.Width(),ixp, ixd);
                         rect2.bottom = rect2.top +
                                    MulDiv(rect2.Height(), iyp, iyd);
                 }

                 pItem->Draw(pDC, rect2);
        if(!pDC->IsPrinting())
        {
                 // draw the tracker over the item
                 CRectTracker tracker;
                 SetupTracker(pItem, &tracker);
                 tracker.Draw(pDC);
                 }
         }
}
```

Additional reference words: kbinf 1.5 1.51 1.52 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.00 4.10
KBCategory: kbprg kbole kbcode kbhowto
KBSubcategory: MfcOLE

How to Insert Objects Without Using Insert Object Dialog Box

PSS ID Number: Q137357

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, and 4.1

SUMMARY

When building an OLE container or OLE server application using MFC OLE classes, you should insert an OLE embedded object programmatically, without using the InsertObject dialog box. This article show you how.

In a default MFC AppWizard-generated OLE container or OLE server application, a command handler is implemented to enable the user to insert an object by clicking Insert New Object on the Edit menu. The AppWizard-generated code makes use of the COleInsertDialog class, which is an MFC wrapper for the OLEUIINSERTOBJECT common dialog box. The COleInsertDialog data and member functions are used to embed the object.

MORE INFORMATION

The member function most responsible for the embedding of an OLE object is COleInsertDialog::CreateItem, which takes a pointer to a COleClientItem as a formal parameter.

When the user clicks Insert New Object on the Edit menu, a COleInsertDialog is created, and it is shown by calling its DoModal function. When the dialog box is dismissed, some of its data members are set by selections made by the user, such as Create From File or Create New.

The implementation of COleInsertDialog::CreateItem calls the COleClientItem member functions to embed the object, which leads to the solution of by-passing the COleInsertDialog class and calling COleClientItem to do the work.

Here is an excerpt from COleInsertDialog::CreateItem:

```
====== Begin Excerpt======
switch (selType)
{
  case linkToFile:
   // link to file selected
  ASSERT(m_szFileName[0] != 0);
        bResult=pNewItem->CreateLinkFromFile(m_szFileName);
  break;
```

```
  case insertFromFile:
  // insert file selected
  ASSERT(m_szFileName[0] != 0);
  bResult=pNewItem->CreateFromFile(m_szFileName);
  break;

  default:
  // otherwise must be create new
  ASSERT(selType == createNewItem);
  bResult=pNewItem->CreateNewItem(m_io.clsid);
  break;
}
```

====== End Excerpt======

This code features a switch structure whose logic flow is controlled by the selType set by the user interaction with the COleInsertDialog dialog box.

The following code demonstrates how to insert an OLE embedded object programmatically. The code shows the creation of an instance of a COleClientItem object, which then calls its CreateNewItem member function to create and embed a Microsoft Excel worksheet.

Sample Code

```
/* Compile options needed : None
*/


void CMyView::OnInsertObject()
{

  BeginWaitCursor();

  CMyOleClientItem* pItem = NULL;
  TRY
  {
      // Create new item connected to this document.
      CMyDoc* pDoc = GetDocument();
      ASSERT_VALID(pDoc);
      pItem = new CMyOleClientItem(pDoc);
      ASSERT_VALID(pItem);

      // Get Class ID for Excel sheet
      // This is used in creation
      CLSID clsid;
      if(FAILED(::CLSIDFromProgID("Excel.Sheet",&clsid)))
        AfxThrowMemoryException();

      // Create the Excel embedded item
      if(!pItem->CreateNewItem(clsid))
        AfxThrowMemoryException();  // any exception will do
      ASSERT_VALID(pItem);
```

```
      // Launch the server to edit the item.
      pItem->DoVerb(OLEIVERB_SHOW, this);

      ASSERT_VALID(pItem);

      // As an arbitrary user interface design, this sets the
      // selection to the last item inserted.

      // TODO: reimplement selection as appropriate for your
      // application

      m_pSelection = pItem;   // set selection to last inserted item
      pDoc->UpdateAllViews(NULL);
  }
  CATCH(CException, e)
  {
      if (pItem != NULL)
      {
        ASSERT_VALID(pItem);
        pItem->Delete();
      }
      AfxMessageBox(IDP_FAILED_TO_CREATE);
  }
  END_CATCH

  EndWaitCursor();
}
```

Note: When using Visual C++ versions 4.0 and 4.1, you need to typecast the 'Excel.Sheet' argument in the previous code sample with the T2COLE macro. In order to use this macro, you need to #include "afxpriv.h" in the view class's implementation file. Also, add the macro 'USES_CONVERSION' in the try block before the following line (which shows the T2COLE macro added):

```
   if (FAILED(::CLSIDFromProgID(T2COLE("Excel.Sheet"),&clsid)))
      AfxThrowMemoryException();
```

Additional reference words: kbinf 1.50 1.51 1.52 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.00 4.10
KBCategory: kbole kbprg kbcode
KBSubcategory: MfcOLE

How to Obtain Filename and Path from a Shell Link or Shortcut

PSS ID Number: Q130698

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.51 and 4.0

SUMMARY
=======

The shortcuts used in Microsoft Windows 95 provide applications and users a way to create shortcuts or links to objects in the shell's namespace. The IShellLink OLE Interface can be used to obtain the path and filename from the shortcut, among other things.

MORE INFORMATION
================

A shortcut allows the user or an application to access an object from anywhere in the namespace. Shortcuts to objects are stored as binary files. These files contain information such as the path to the object, working directory, the path of the icon used to display the object, the description string, and so on.

Given a shortcut, applications can use the IShellLink interface and its functions to obtain all the pertinent information about that object. The IShellLink interface supports functions such as GetPath(), GetDescription(), Resolve(), GetWorkingDirectory(), and so on.

Sample Code

The following code shows how to obtain the filename or path and description of a given link file:

```
#include <windows.h>
#include <shlobj.h>

// GetLinkInfo() fills the filename and path buffer
// with relevant information.
// hWnd         - calling application's window handle.
//
// lpszLinkName - name of the link file passed into the function.
//
// lpszPath     - the buffer that receives the file's path name.
//
// lpszDescription - the buffer that receives the file's
// description.
HRESULT
GetLinkInfo( HWND    hWnd,
             LPCTSTR lpszLinkName,
             LPSTR   lpszPath,
             LPSTR   lpszDescription)
```

```
{

   HRESULT hres;
   IShellLink *pShLink;
   WIN32_FIND_DATA wfd;

// Initialize the return parameters to null strings.
   *lpszPath = '\0';
   *lpszDescription = '\0';

// Call CoCreateInstance to obtain the IShellLink
// Interface pointer. This call fails if
// CoInitialize is not called, so it is assumed that
// CoInitialize has been called.
   hres = CoCreateInstance( &CLSID_ShellLink,
                            NULL,
                            CLSCTX_INPROC_SERVER,
                            &IID_IShellLink,
                            (LPVOID *)&pShLink );

   if (SUCCEEDED(hres))
   {
      IPersistFile *ppf;

// The IShellLink Interface supports the IPersistFile
// interface. Get an interface pointer to it.
      hres = pShLink->lpVtbl->QueryInterface(pShLink,
                                        &IID_IPersistFile,
                                        (LPVOID *)&ppf );
      if (SUCCEEDED(hres))
      {
         WORD wsz[MAX_PATH];

// Convert the given link name string to a wide character string.
         MultiByteToWideChar( CP_ACP, 0,
                              lpszLinkName,
                              -1, wsz, MAX_PATH );
// Load the file.
         hres = ppf->lpVtbl->Load(ppf, wsz, STGM_READ );
         if (SUCCEEDED(hres))
         {
// Resolve the link by calling the Resolve() interface function.
// This enables us to find the file the link points to even if
// it has been moved or renamed.
            hres = pShLink->lpVtbl->Resolve(pShLink,  hWnd,
                                            SLR_ANY_MATCH | SLR_NO_UI);
            if (SUCCEEDED(hres))
            {
// Get the path of the file the link points to.
               hres = pShLink->lpVtbl->GetPath( pShLink, lpszPath,
                                            MAX_PATH,
                                            &wfd,
                                            SLGP_SHORTPATH );

// Only get the description if we successfully got the path
```

```
   // (We can't return immediately because we need to release ppf &
   //  pShLink.)
                  if(SUCCEEDED(hres))
                  {
   // Get the description of the link.
                    hres = pShLink->lpVtbl->GetDescription(pShLink,
                                                           lpszDescription,
                                                           MAX_PATH );
                  }
               }
            }
            ppf->lpVtbl->Release(ppf);
         }
         pShLink->lpVtbl->Release(pShLink);
      }
      return hres;
   }
```

Additional reference words: 4.00
KBCategory: kbprg kbcode kbole
KBSubcategory: IShellLink

How to Optimize the Reactivation of In-Place Active Servers

PSS ID Number: Q137139

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52, 1.52b
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY
=======

When an OLE container embeds an in-place active capable server, the deactivation/reactivation process may be considered too long. One way to optimize the process is to hide the server rather than deactivate it. Simply hiding the server will enable a faster reactivation should the embedded object be reactivated.

MORE INFORMATION
================

Optimizing the deactivation/reactivation process for an object embedded in an MFC container application involves overriding the virtual COleClientItem::OnDeactivateUI and COleClientItem::OnActivate methods.

By default, MFC OLE container applications created with versions of AppWizard prior to version 4.0 contain the following implementation of the COleClientItem::OnDeactivateUI method:

```
   void CCntrItem::OnDeactivateUI(BOOL bUndoable)
   {
     COleClientItem::OnDeactivateUI(bUndoable);

     // Close an in-place active item whenever it removes the user
     // interface. The action here should match as closely as possible
     // the handling of the ESC key in the view.

     Deactivate();   // nothing fancy here -- just deactivate the object
   }
```

To optimize the deactivation/reactivation process, modify the AppWizard generated version of this function to have it invoke the embedded item's OLEIVERB_HIDE verb to have the item hide itself:

```
   void CCntrItem::OnDeactivateUI(BOOL bUndoable)
   {
     COleClientItem::OnDeactivateUI(bUndoable);

     // Hide the object if it is not an outside-in object
     DWORD dwMisc = 0;
     m_lpObject->GetMiscStatus(GetDrawAspect(), &dwMisc);
     if (dwMisc & OLEMISC_INSIDEOUT)
```

```
        DoVerb(OLEIVERB_HIDE, NULL);
   }
```

The second step in optimizing the deactivation/reactivation process involves modifying the virtual COleClientItem::OnActivate method to correctly handle activation of another embedded object. To override COleClientItem::OnActivate, first add the following public method declaration to the declaration of the MFC container application's COleClientItem derived class:

```
   virtual void OnActivate();
```

Add the following definition of the OnActivate method to the implementation file for the container's COleClientItem derived class:

```
   void CCntrItem::OnActivate()
   {
     // allow only one in-place active item per frame
     CView* pView = (CView*)GetActiveView();
     ASSERT_VALID(pView);
     COleClientItem* pItem = GetDocument()->GetInPlaceActiveItem(pView);
     if (pItem != NULL && pItem != this)
       pItem->Close();

     COleClientItem::OnActivate();
   }
```

If the container has previously hidden an embedded object, this function will close it prior to activating the new item. Note that this optimization is similar to the technique that Microsoft Excel and the rich edit control use in similar circumstances.

Beginning with Visual C++ version 4.0, AppWizard-generated OLE containers contain the optimized versions of COleClientItem::OnDeactivateUI and COleClientItem::OnActivate automatically. There is no need to manually add the changes shown above.

Additional reference words: kbinf 1.50 1.51 1.52 1.52b 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.00 Word 6.0 6.0c 95 embedded UIActive
KBCategory: kbole kbcode
KBSubcategory: MfcOLE

How to Override an Interface in an MFC Application

PSS ID Number: Q141277

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY

In an MFC application, you can override existing interfaces in a class as well as provide additional interfaces. Overriding an interface in this case is synonymous with replacing an interface. The example in this article illustrates how to override an interface in a class while preserving the original interface implementation so that it can be delegated to by the new interface implementation.

This article doesn't deal with overriding the IDispatch implementation as this is a special case. The following article demonstrates how to override IDispatch in MFC:

   ARTICLE-ID: Q140616
   TITLE   : MFCDISP: Replacing MFC's IDispatch implementation

MORE INFORMATION

The following steps will override the IOleObject implementation for a default Ole Control generated by the Control Wizard.

1. To add the declaration of the IOleObject implementation to the control, add the following code to the header file for the COleControl-derived class:

```
// Interface Maps
protected:
     // IOleObject
     BEGIN_INTERFACE_PART(MyOleObject, IOleObject)
         INIT_INTERFACE_PART(CIOleOverCtrl, MyOleObject)
         STDMETHOD(SetClientSite)(LPOLECLIENTSITE);
         STDMETHOD(GetClientSite)(LPOLECLIENTSITE*);
         STDMETHOD(SetHostNames)(LPCOLESTR, LPCOLESTR);
         STDMETHOD(Close)(DWORD);
         STDMETHOD(SetMoniker)(DWORD, LPMONIKER);
         STDMETHOD(GetMoniker)(DWORD, DWORD, LPMONIKER*);
         STDMETHOD(InitFromData)(LPDATAOBJECT, BOOL, DWORD);
         STDMETHOD(GetClipboardData)(DWORD, LPDATAOBJECT*);
         STDMETHOD(DoVerb)(LONG, LPMSG, LPOLECLIENTSITE, LONG, HWND,
                 LPCRECT);
         STDMETHOD(EnumVerbs)(IEnumOLEVERB**);
         STDMETHOD(Update)();
         STDMETHOD(IsUpToDate)();
```

```
            STDMETHOD(GetUserClassID)(CLSID*);
            STDMETHOD(GetUserType)(DWORD, LPOLESTR*);
            STDMETHOD(SetExtent)(DWORD, LPSIZEL);
            STDMETHOD(GetExtent)(DWORD, LPSIZEL);
            STDMETHOD(Advise)(LPADVISESINK, LPDWORD);
            STDMETHOD(Unadvise)(DWORD);
            STDMETHOD(EnumAdvise)(LPENUMSTATDATA*);
            STDMETHOD(GetMiscStatus)(DWORD, LPDWORD);
            STDMETHOD(SetColorScheme)(LPLOGPALETTE);
        END_INTERFACE_PART(MyOleObject)

    DECLARE_INTERFACE_MAP();
```

This adds a nested class XMyOleObject to your control class. Note these macros declare interface methods including the IUnknown interface methods, so you must implement the IUnknown methods as well.

2. Add the IOleObject interface to the interface map for the control by adding an INTERFACE_PART macro to the implementation file for the control:

```
BEGIN_INTERFACE_MAP(CIOleOverCtrl, COleControl)
   INTERFACE_PART(CIOleOverCtrl, IID_IOleObject, MyOleObject)
END_INTERFACE_MAP()
```

Replace CIOleOverCtrl with the name of your control and MyOleObject with the name you chose for the nested class that supports IOleObject.

3. Implement the interface methods you declared. Add the following code to the implementation file for the control:

```
STDMETHODIMP_(ULONG) CIOleOverCtrl::XMyOleObject::AddRef()
{
    METHOD_MANAGE_STATE(CIOleOverCtrl, MyOleObject)
    ASSERT_VALID(pThis);

    return pThis->m_xOleObject.AddRef();
}

STDMETHODIMP_(ULONG) CIOleOverCtrl::XMyOleObject::Release()
{
    METHOD_MANAGE_STATE(CIOleOverCtrl, MyOleObject)
    ASSERT_VALID(pThis);

    return pThis->m_xOleObject.Release ();
}

STDMETHODIMP CIOleOverCtrl::XMyOleObject::QueryInterface(
    REFIID iid, LPVOID far* ppvObj)
{
    METHOD_MANAGE_STATE(CIOleOverCtrl, MyOleObject)
    ASSERT_VALID(pThis);

    return pThis->m_xOleObject.QueryInterface ( iid,  ppvObj);
}
```

```
STDMETHODIMP
CIOleOverCtrl::XMyOleObject::SetClientSite(LPOLECLIENTSITE
pClientSite)
{
    METHOD_MANAGE_STATE(CIOleOverCtrl, MyOleObject)
       ASSERT_VALID(pThis);

    return pThis->m_xOleObject.SetClientSite ( pClientSite );
}
...
```

The rest of the methods follow the same pattern where CIOleOverCtrl is the name of the control, XMyOleObject is the name of the nested class that supports IOleObject, and m_xOleObject is calculated by removing the I from the interface being supported and adding m_x.

Note that these methods simply pass the call on to the original IOleObject implementation. However, this is not a requirement; you could add functionality and delegate to the original implementation or not delegate at all.

REFERENCES
==========

Technical Notes #38 and #39.

Additional reference words: 3.00 3.10 3.20 4.00
KBCategory: kbole kbprg kbhowto kbocde
KBSubcategory: MfcOle

How to Pass IDispatch Pointer & Avoid an Application Error

PSS ID Number: Q133042

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2,
  4.0, and 4.1

SUMMARY

In some cases, you may want an OLE Automation Local-Server to pass its IDispatch pointer to another Local-Server through an automation method. However, the pointer will become invalid after the method in the second server returns, so an Access Violation will occur if the second server tries to use the pointer later.

To maintain the integrety of the pointer, you need to call AddRef on the IDispatch pointer within the second server, as described in the following rule from the OLE SDK documentation:

> If a local copy of an interface pointer is made from an existing global interface pointer, the local copy must be independently reference-counted. This separate reference count is necessary because the global copy can be destroyed while the local copy is in use.

MORE INFORMATION

Here is an example demonstrating the reasoning behind the general rule.

When the IDispatch pointer from the first Local-Server (server1) is passed to the second Local-Server (server2), the RPC manager creates a proxy in the address space of server2. The pointer that server2 actually receives points to the proxy. This should make sense because the original object resides in the address space of server1, so server2 cannot actually have a pointer to it.

The default life-time of the proxy is the duration of the function or automation method. In many cases, however, it would be convenient to use the pointer later. In MFC, you can do this by creating an instance of COleDispatchDriver (or an instance of an object derived from COleDispatchDriver) and calling AttachDispatch().

To affect the life-time of the proxy, server2 must call AddRef for the automation object it receives. The RPC Manager will intercept this call and increment the reference count on both the proxy and the object. It is important to note that AddRef must be called from server2 because the proxy lives in server2's address space.

The following sample code demonstrates how to do it. It is assumed that CServer2Doc is an automation object. The CServerDoc2::SetReturnDispatch

method receives a LPDISPATCH object from server1 and will attach it to CServer2Doc::m_DDServer1, which is a instance of an object derived from COleDispatchDriver. Note that m_DDServer1 will be deleted when CServer2Doc is deleted; Because AttachDispatch is called with the default second parameter, this will cause the LPDISPATCH object Release to be called.

Refer to COleDispatchDriver::~COleDispatchDriver and COleDispatchDriver::AttachDispatch for more information.

Sample Code

```
void CServer2Doc::SetReturnDispatch(LPDISPATCH lpDispServer1)
{
  lpDispServer1->AddRef();         // AddRef so it can be used later
  m_DDServer1.AttachDispatch(lpDispServer1);
}
```

REFERENCES
==========

- Visual C++ Books Online Contents\OLE 2.0 SDK\Chapter 6 Component Object Interfaces and Functions \Iunknown Interface\Reference Counting Rules.

- OLE 2 Programmers Reference Volume One, pages 191-195.

- Inside OLE 2, pages 83-90.

Additional reference words: kbinf 2.00 3.0 3.00 2.10 3.1 3.10 2.20 3.2 3.20 4.00 4.10
KBCategory: kbole kbcode
KBSubcategory: MfcOLE

How to Profile an OLE Server or an OLE Control

PSS ID Number: Q147393

The information in this article applies to:

- The Microsoft Source Profiler included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SUMMARY

The Visual C++ Books Online section "Profiling Dynamic-Link Libraries" explains the steps involved in profiling dynamic-link libraries and profiling multiple .dll and .exe files.

Because an OLE server is basically an .exe or .dll file and an OLE control is a DLL even though it has an extension of .ocx, profiling them is similar to profiling an .exe or .dll file. When you profile an OLE server or OLE control, you follow the same steps and use the same commands that you would for an .exe or .dll file. The same profiling rules that apply to .exe and .dll files (such as the need to include debugging information and to supply a .map file) also apply to OLE servers and OLE controls.

MORE INFORMATION

The following four examples illustrate the steps required to profile an OLE full server, an OLE mini-server, an In-Process OLE Automation server, and an OLE control.

You may need to register the OLE mini-server, the in-process OLE automation server, or the OLE control explicitly. You can do one of the following to register them:

- For a .dll file (and for an OLE control), run Regsvr32 from the command prompt. Alternatively, you can open the corresponding project in the Developer Studio, and then on the Tools menu, click Register Control.

- For an .exe file, just run it once. You may not be able to run it or you may get a message saying "This server can only be run from a container application," but when you click OK, it will have been registered.

To run the profiler, you may need to copy the \Msdev\Bin\Profile.dll file to either the directory that contains the file you are profiling or to one of the system directories.

Each step in the following examples is a command at the command prompt. Alternatively, you could create a batch file that contains these commands.

In the command line arguments for most profile commands, you should specify the file names without the extension. Look at the following examples to see when extensions should be included.

To profile an OLE full server or an OLE automation server:

```
PREP /OM /FT FullSrvr
PROFILE FullSrvr
PREP /M FullSrvr
PLIST FullSrvr > FullSrvr.txt
```

where FullSrvr.exe is the file corresponding to an OLE full server or OLE automation server.

To profile an OLE mini server:

```
COPY MiniSrvr.exe MiniSrvr.sav
PREP /OM /FT MiniSrvr
COPY MiniSrvr._xe MiniSrvr.exe
PROFILE /I MiniSrvr /O MiniSrvr Container
PREP /M MiniSrvr
PLIST MiniSrvr > MiniSrvr.txt
COPY MiniSrvr.sav MiniSrvr.exe
```

where MiniSrvr.exe is the file corresponding to an OLE mini server and Container.exe is a container application that runs the MiniSrvr.

To profile an In-Process OLE Automation Server (these steps are similar to those of OLE mini server except that you have a .dll extension instead of an .exe extension):

```
COPY InProc.dll InProc.sav
PREP /OM /FT InProc.dll
COPY InProc._ll InProc.dll
PROFILE /I InProc /O InProc Client
PREP /M InProc
PLIST InProc > InProc.txt
COPY InProc.sav InProc.dll
```

where InProc.dll is the file corresponding to an In-Process OLE Automation server and Client is an OLE Automation client application that drives the InProc.dll.

To profile an OLE control that has an .ocx extension (these steps are also similar except that you now have an .ocx extension):

```
COPY Control.ocx Control.sav
PREP /OM /FT Control.ocx
COPY Control._cx Control.ocx
PROFILE /I Control /O Control ControlContainer
PREP /M Control
PLIST Control > Control.txt
COPY Control.sav Control.ocx
```

where Control.ocx is the OLE control and ControlContainer is an OLE control container application.

REFERENCES
==========

Profiler Reference, Visual C++ User's Guide.

Advanced Profiling, Programming Techniques.

Both of these references are available in the online documentation that comes with Visual C++ version 4.0.

Additional reference words: kbinf 4.00 mini-server
KBCategory: kbtool kbusage kbole kbhowto kbcode
KBSubcategory: VWBIss

How to Register an MFC OLE Automation Server to Avoid Failure

PSS ID Number: Q137516

The information in this article applies to:

The AppWizard, included with:

- Microsoft Visual C++, 32-bit Edition, version(s) 2.00 2.10 2.20

SUMMARY

Creation of an OLE Object may fail if the registry file (.reg) generated by AppWizard is used to register the server for the object. This article explains how to avoid this failure.

MORE INFORMATION

Registration of an MFC OLE Automation server typically occurs when running the server stand-alone for the first time. An alternate way of registering the server is to run the registration editor and merge the .reg file generated by AppWizard for the server's project. This method is necessary if the server is installed but not run before a client tries to create it.

It is necessary that the setup program that installs the server also replace any occurrence of the executable in the .reg file with the fully-qualified path and file name of the executable. When AppWizard creates the .reg file for the project, AppWizard does not know what the final path to the executable will be. Therefore, AppWizard cannot fill in the fully qualified path name for the server.

If the .reg file provided by the AppWizard is used as is to register the server for an object, OLE won't be able to find the server, creation of the object will fail, and OLE will return the error CO_E_SERVER_EXEC_FAILURE.

Sample Code

The following is an example of the \LocalServer key for the AutoClik, Step 1 sample as generated by AppWizard:

```
HKEY_CLASSES_ROOT\CLSID\{2106E720-AEF8-101A-
     9005-00DD0108D651}\LocalServer = AUTOCLIK.EXE
```

After being setup correctly, the \LocalServer key in the .reg file may appear as follows:

```
HKEY_CLASSES_ROOT\CLSID\{2106E720-AEF8-101A-
     9005-00DD0108D651}\LocalServer = C:\MSVC20\SAMPLES\MFC
                          \autoclik\step3\WinDebug\AUTOCLIK.EXE
```

Additional reference words: kbinf 2.00 2.10 2.20 3.00 3.10 3.20
KBCategory: kbole kbenv kbtshoot
KBSubcategory:

How To Retrieve Dialog Info from Word Using an MFC App

PSS ID Number: Q152070

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SUMMARY
=======

This article describes how to retrieve values from Word's dialog boxes using OLE Automation in an MFC-based application. To retrieve values, the client application must create a CurValues object and an object that will represent the data in one of Word's dialogs.

MORE INFORMATION
================

Using Visual C++, it is possible to create a wrapper class for a Word type library. This wrapper class represents a WordBasic object. Using the WordBasic class, it is possible to call many methods of the WordBasic object from within an MFC application. However, it is not possible to get the settings of any of the Word dialogs using the WordBasic object's methods directly. You can get information such as Summary Info and Word Count from Word's dialogs by using a CurValues object. To accomplish this in an MFC application, you must take the following steps:

1. Use ClassWizard to create an MFC-based application to serve as the Automation client. Ensure that OLE has been properly initialized for your MFC application. This is usually done by calling AfxOleInit() in the CWinApp-derived class's InitInstance. This example also assumes that you have correctly installed Word 6.0 or 7.0 on your system and that it is registered correctly with the system registry.

2. Create a WordBasic class by using Class Wizard to create a new class from a type library. Class libraries available for Word automation are:

   - wb60en.tlb for Word 6.0

   - wb70en32.tlb for Word 7.0.

3. In your code somewhere, instantiate a WordBasic object and attach it to a COleDispatchDriver. The following code shows how to do this for the WordBasic class created by Class Wizard from one of the Word type libraries:

   ```
   WordBasic wb;
   wb.CreateDispatch("Word.Basic");
   ```

4. Use the WordBasic object to get the DISPID for its CurValues object. For example, the following code gets the DISPID for the CurValues object using the WordBasic object from step 2:

```
OLECHAR* lpszCurValues = L"CurValues";
DISPID dispidCurValues;
wb.m_lpDispatch->GetIDsOfNames( IID_NULL, &lpszCurValues, 1,
                                LOCALE_SYSTEM_DEFAULT, &dispidCurValues )
```

5. Use the CurValues DISPID to get the CurValues property of the WordBasic object. This property is of type VT_DISPATCH and can be attached to a COleDispatchDriver as shown below:

```
COleDispatchDriver dispdrvrCurValues;
LPDISPATCH resultDispatch;
wb.GetProperty(dispidCurValues, VT_DISPATCH,
                 (void*)&resultDispatch);
dispdrvrCurValues.AttachDispatch(resultDispatch);
```

6. Use the COleDispatchDriver from step 4 to get the DISPID for the dialog whose information you wish to retrieve. For example, to get the document's Summary Info, use code similar to the following:

```
OLECHAR* lpszFileSummaryInfo = L"FileSummaryInfo";
DISPID dispidFileSummaryInfo;

dispdrvrCurValues.m_lpDispatch->GetIDsOfNames( IID_NULL,
&lpszFileSummaryInfo, 1, LOCALE_SYSTEM_DEFAULT,&dispidFileSummaryInfo)
```

7. Use the CurValues object to get a property of type VT_DISPATCH that represents the dialog containing the information you wish to retrieve. The LPDISPATCH you retrieve can be attached to a COleDispatchDriver as shown below:

```
COleDispatchDriver dispdrvrFileSummaryInfo;
LPDISPATCH resultDispatch;

dispdrvrCurValues.GetProperty(dispidFileSummaryInfo, VT_DISPATCH,
   (void*)&resultDispatch);
dispdrvrFileSummaryInfo.AttachDispatch(resultDispatch);
```

8. You can now use the COleDispatchDriver from step 6 to get values from the dialog. To get the document title from the Summary Info dialog from step 6, you would use code similar to the following:

```
OLECHAR* lpszTitle = L"Title";
DISPID dispidDocProperty;
BSTR *pResult;

dispdrvrFileSummaryInfo.m_lpDispatch->GetIDsOfNames( IID_NULL,
    &lpszTitle, 1, LOCALE_SYSTEM_DEFAULT, &dispidDocProperty);

dispdrvrFileSummaryInfo.GetProperty(dispidDocProperty,VT_BSTR,pResult);
```

REFERENCES
==========

WordBasic Help, shipped with Microsoft Word version 6.0 and 7.0.

Additional reference words: 2.00 2.10 2.20 4.00 4.10
KBCategory: kbprg kbole kbappnote kbhowto
KBSubcategory: MfcOLE VCx86

How to Retrieve the Actual Parent Window of an OLE Control

PSS ID Number: Q150204

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

  - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SUMMARY
=======

An OLE control is created as a child of the window of the container most closely associated with the site object. This article describes how to obtain the Parent window of an OLE control, and why calling GetParent or CWnd::GetParent for an OLE control may not return its actual Parent window.

MORE INFORMATION
================

An OLE control typically gains access to the following container windows:

- Frame window: the outer-most container window where the container's main menu resides. An OLE control retrieves this window handle by calling either IOleInPlaceFrame::GetWindow or IOleInPlaceSite::GetWindowContext.

- Site window: the container window that contains the OLE control's view. An OLE control retrieves this container window by calling IOleInPlaceSite::GetWindow.

However, OLE control containers generated using MFC and Microsoft Visual Basic 4.0 use the same window for the frame and the site object.

When the MFC framework creates an OLE control's window, it retrieves the window associated with the container's site object by calling IOleInPlaceSite::GetWindow. The returned window is made the parent of the OLE control, except in the following two cases, where the framework creates a reflector window that reflects the notification messages:

- The OLE control subclasses a Windows control, and the container does not support message reflection.

  -or-

- The control container does not support automatic clipping of its controls.

In these two scenarios, the OLE control is made a child of the reflector window whose parent is the window returned from IOleInPlaceSite::GetWindow.

NOTE: If IOleObject::DoVerb with OLEIVERB_OPEN is invoked on an OLE

control, and if in-place activation is not possible, an outer frame window
is created and becomes the parent of the OLE control.

Sample Code

```
// The following code should return the
// actual parent window of the OLE control
HWND CMyOleControl::GetActualParent()
{
    HWND hwndParent = 0;

    // Get the window associated with the in-place site object,
    // which is connected to this OLE control
    if (m_pInPlaceSite != NULL)
        m_pInPlaceSite->GetWindow(&hwndParent);

    return hwndParent;     // return the in-place site window handle
}
```

REFERENCES
==========

- Online Documentation for IOleWindow::GetWindow

- MFC source code for COleControl::OnActivateInPlace and
  COleControl::CreateControlWindow.

Additional reference words: 1.50 2.00 2.10 2.20 4.00 4.10 kbinf ocx control
KBCategory: kbprg kbole kbhowto
KBSubcategory: MfcOLE CDKIss

How to Set Default Values for Stock Properties in OLE Controls

PSS ID Number: Q138866

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, and 4.1

SUMMARY

When developing an OLE Control, you may decide that some of the properties should be serialized. This can be accomplished in the DoPropExchange() method of your COleControl-derived class by using the PX_ functions. The PX_ functions also allow you to specify a default value for the property, which is good programming practice. You may also want to give default values to any stock properties that are used by your control. The serialization and initialization of stock properties is handled by MFC in COleControl::DoPropExchange(pPX). To set a default value for a stock property, you need to work around some of MFC's functionality. This article explains how to do it.

MORE INFORMATION

When you use the Microsoft Control Developer's Kit (CDK) to create the MFC framework for an OLE Control, it creates a COleControl-derived class with an overridden DoPropExchange() member function. By default, the overridden DoPropExchange() calls COleControl::DoPropExchange(). This in turn calls COleControl::ExchangeStockProps(), which contains the MFC code for serializing stock properties.

Within this function, MFC uses the COleControl m_dwStockPropMask protected member variable to determine which stock properties you have added with Class Wizard. This mask can be modified in your DoPropExchange() to force MFC to ignore existing stock properties and therefore allow you to add your own code. Ideally you should copy the relevant code already present in COleControl::ExchangeStockProps(), and then modify it as necessary.

The following sample code shows how to accomplish this for the Caption stock property by modifying step three of the Circ OLE Control tutorial.

Sample Code

```
// MANUAL CONTROL OF MAKING Caption PERSISTANT.
// To test the initialization of the caption stock property
// replace the circle3 samples DoPropExchange function with the
// following version.

// These #defines can be found in the ..\Msvc20\Cdk32\Src\Ctlprop.cpp
// file and are given here to remove the need to continually track them
// down. In the code, only STOCKPROP_CAPTION is used.
```

```
#define STOCKPROP_BACKCOLOR     0x00000001
#define STOCKPROP_CAPTION       0x00000002
#define STOCKPROP_FONT          0x00000004
#define STOCKPROP_FORECOLOR     0x00000008
#define STOCKPROP_TEXT          0x00000010
#define STOCKPROP_BORDERSTYLE   0x00000020
#define STOCKPROP_ENABLED       0x00000040

void CCirc3Ctrl::DoPropExchange(CPropExchange* pPX)
{
   ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
   BOOL bLoading = pPX->IsLoading();

   CString strText;
   if (!bLoading)
      strText = InternalGetText();

   PX_String(pPX,_T("Caption"),strText,_T("Caption"));
   if (bLoading) {
      TRY
         SetText(strText);
      END_TRY
   }

   // Mask out the Caption stock property before
   // calling the base class functionality.
   m_dwStockPropMask &= ~STOCKPROP_CAPTION;

   // Call the base class.
   COleControl::DoPropExchange(pPX);

   // Mask the Caption property bit back in to minimize
   // possible side effects.
   m_dwStockPropMask |= STOCKPROP_CAPTION;

   if (pPX->GetVersion() == (DWORD)MAKELONG(_wVerMinor, _wVerMajor))
   {
      PX_Bool(pPX, _T("CircleShape"), m_circleShape, TRUE);
      PX_Short(pPX, _T("CircleOffset"), m_circleOffset, 0);
      PX_Long(pPX, _T("FlashColor"), (long &)m_flashColor,
                     RGB(0xFF, 0x00, 0x00));
      PX_String(pPX, _T("Note"), m_note, _T("Note"));
   }else if (pPX->IsLoading()) {
      m_circleShape = TRUE;
      m_circleOffset = 0;
      m_flashColor = RGB(0xFF, 0x00, 0x00);
      m_note = _T("");
   }
}
```

REFERENCES
==========

The functions discussed in this article can be found using the Find in Files feature of Visual C++; in version 2.x within the Cdk32\Src directory

and in version 4.x within the mfc\src subdirectory.

Additional reference words: 2.00 2.10 2.20 3.0 3.1 3.2 3.00 3.10 3.20
4.00 4.10 cdk ocx
KBCategory: kbprg kbole kbhowto kbcode
KBSubcategory: MfcOLE

How to Set the Picture Property of an OLE Control

PSS ID Number: Q146010

The information in this article applies to:

- Microsoft Visual C++ 32-bit Edition, version 4.0

SUMMARY
=======

To set the Picture property of an OLE control, you can use the Component Gallery to insert an OLE control into an AppWizard-generated application with control container support. If the control has a picture property, the wrapper classes generated by the Component Gallery will contain the SetPicture() and GetPicture() methods. The More Information section below contains details that explain how to use the picture property of an OLE Control.

MORE INFORMATION
================

Using the Microsoft Grid Control as an example, you can implement the SetPicture() and GetPicture() methods in the control classes provided by the Component Gallery by using this code:

```
void CGridCtrl::SetPicture(LPDISPATCH propVal)
{
     SetProperty(0x15, VT_DISPATCH, propVal);
}

CPicture CGridCtrl::GetPicture()
{
     LPDISPATCH pDispatch;
     GetProperty(0x15, VT_DISPATCH, (void*)&pDispatch);
     return CPicture(pDispatch);
}
```

It is not intuitively clear how to use these methods to get or set the picture property. The following steps show how to set the picture property of an OLE control successfully in an AppWizard-generated application. The particular control used in this example is the Microsoft Grid Control, and its picture property is being set to the toolbar bitmap provided by AppWizard.

1. Generate a new AppWizard application. Be sure to select both Container and Control support in AppWizard Step 3.

2. Add the Grid Control from the OLE Controls tab in the Component Gallery.

3. Add Afxctl.h to the list of pre-compiled header files in Stdafx.h.

4. Add a Grid Control object to the About Dialog box in the dialog editor.

5. Using Class Wizard, add a control member variable called m_grid to the Grid Control in the About Dialog box.

6. Override the container's initialization routine to create and initialize a CPictureHolder object. Call the SetPicture() method for the control, passing in the CPictureHolder's dispatch pointer. SetPicture calls SetProperty() for the control. Here's an example:

```
BOOL CAboutDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Create and Initialize the CPictureHolder variable with the
    // toolbar resource
    CPictureHolder pictholder;
    pictholder.CreateFromBitmap(IDR_MAINFRAME);

    //Pass dispatch pointer to CPictureHolder
    m_grid.SetPicture(pictholder.GetPictureDispatch());

    // NOTE: GetPictureDispatch() QI()'s for the IPictureDisp
    // interface, so the picture object will remain alive until
    // the property is reset or the control is destroyed. When
    // pictholder goes out of scope, it calls Release() on the
    // picture object, but the previously mentioned QI() will have
    // bumped the ref count, allowing it to remain alive until the
    // control itself releases the picture object.

  return TRUE;
}
```

REFERENCES
==========

OLE Control Containers: Programming OLE Controls in an OLE Control Container - Visual C++ Books Online, MFC Encyclopedia.

OLE Controls: Using Pictures in an OLE Control - Visual C++ Books Online, MFC Encyclopedia.

Additional reference words: kbinf 4.00
KBCategory: kbole kbcode kbhowto
KBSubcategory: CDKIss MfcOLE

How to Show Container's Toolbar During Inplace Activation

PSS ID Number: Q148860

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SUMMARY

This article illustrates how to make an OLE Container's toolbar visible when the user is editing an inplace active object. The technique makes both the container and the server toolbars visible during inplace editing.

MORE INFORMATION

During inplace activation, MFC shows and hides control bars that have the CBRS_HIDE_INPLACE style used in the COleClientItem::OnShowControlBars() function, which can be found in the MFC source code (Olecli2.cpp).

MFC automatically adds the CBRS_HIDE_INPLACE style to toolbars that are created with the ID AFX_IDW_TOOLBAR, which is the default value for the toolbar created by AppWizard in the MainFrame window (m_wndToolBar). You can see this in the CToolBar::Create() function in the MFC source code (Bartool.cpp). To remove the CBRS_HIDE_INPLACE style from m_wndToolBar, add the following line of code to the CMainFrame::OnCreate function after m_wndToolBar is created:

```
   m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle()
                            ^ CBRS_HIDE_INPLACE);
```

The SetBarStyle() call removes the CBRS_HIDE_INPLACE style by performing a bitwise XOR operation with the existing style for the toolbar.

NOTE: The resulting behavior depends on the server's implementation. For example, Word 6.0 always puts its toolbar at position (0,0). In this case, Word's toolbar will be placed in front of the container's toolbar. Most servers place their toolbar beneath the container's toolbar as expected.

Additional reference words: kbinf 2.00 2.10 2.20 4.00 4.10
KBCategory: kbole kbhowto kbcode
KBSubcategory: MfcOLE

How to Update Property Values in a Property Page

PSS ID Number: Q148222

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft Visual C++, 32-bit Edition, version 4.0
  - Microsoft OLE Control Developer's Kit (CDK) versions 2.0, 2.1, 2.2

SUMMARY

To modify a property of an OLE control, you would normally change the properties value in the control's property page. However, it is possible for a property to be modifiable from more than one property page especially in development environments such as Visual Basic. In this case, when you change the property in the control's property page, you should synchronize the value in Visual Basic's Property Form. This article shows by example how to do it.

MORE INFORMATION

For example, in Visual Basic, place the Circ3 sample control on the Form page. Then click the right mouse button to select properties for the control. You will notice that the same properties can be seen in Visual Basic's Properties window. For example, one of the properties of Circ3 is the CircleOffset property. If you change this value in the Control's property page and click Apply, you will notice that the value has not been updated in Visual Basic's Properties window. To synchronize the value in both, you need to call BoundPropertyChanged(dispid) for each property you want to update.

Note that the dispid is the id of the property to update. It is usually found in the header file of the COleControl-derived class. The dispids for the Circ3 control follow:

```
// Dispatch and event IDs
public:
       enum {
       //{{AFX_DISP_ID(CCirc3Ctrl)
       dispidCircleShape = 2L,
       dispidCircleOffset = 3L,
       dispidFlashColor = 1L,
       dispidNote = 4L,
       eventidClickIn = 1L,
       eventidClickOut = 2L,
       //}}AFX_DISP_ID
       };
```

The following sample code shows how to modify the Circ3 sample to synchronize the CircleShape and CircleOffset properties given the dispids

above.

Sample Code

```
/* Compile options needed: none
*/

void CCirc3Ctrl::OnCircleShapeChanged()
{
   SetModifiedFlag();

   // force the control to redraw itself
   InvalidateControl();

   // reset the circle offset, if necessary
   if (m_circleShape)
      SetCircleOffset(0);

   BoundPropertyChanged(dispidCircleShape);      // *ADD THIS LINE*
}

void CCirc3Ctrl::SetCircleOffset(short nNewValue)
{
   // Validate the specified offset value
   if ((m_circleOffset != nNewValue) && m_circleShape&&
       InBounds(nNewValue))
   {
      m_circleOffset = nNewValue;
      SetModifiedFlag();
      BoundPropertyChanged(dispidCircleOffset); // *ADD THIS LINE*
      InvalidateControl();
   }
}
```

Additional reference words: kbinf 2.00 2.10 2.20 4.00 3.00 3.10 3.20
Property Page Form Browser
KBCategory: kbole kbhowto kbocde
KBSubcategory: CDKIss MfcOLE

How to Use AFX_MANAGE_STATE in an OLE Control

PSS ID Number: Q127074

The information in this article applies to:

- The OLE Control Developer's Kit (CDK), versions 1.0 and 1.1

SUMMARY
=======

The AFX_MANAGE_STATE macro is used to establish the correct module context for OLE controls and should be used for all external entry points to the control.

The existing implementation of COleControl and COlePropertyPage use AFX_MANAGE_STATE in several locations, but there are still some situations where the control writer will need to directly invoke AFX_MANAGE_STATE. This article shows you how.

MORE INFORMATION
================

OLE controls written with the OLE CDK use a shared MFC DLL to eliminate the need to have a separate copy of MFC in each control. In a way, an OLE control is similar to a standard MFC extension DLL. Previous versions of MFC use the _AFXDLL model to implement extension DLLs, one limitation of this model is that the host executable had to be written to also use the shared MFC DLL, that is it must be used by an MFC application. A different mechanism is needed for OLE controls because they can be used by non-MFC applications.

MFC maintains a small amount of global state information that includes (among other things) the instance handle used when loading resources. Many MFC functions rely on this state information, so it is important that it always reflect the state of the module currently executing.

The AFX_MANAGE_STATE macro creates a small local object on the stack that swaps in the control's state information in its constructor, and then restores the previous state in its destructor. A typical use of the macro looks like this:

```
   AFX_MANAGE_STATE(_afxModuleAddrThis);
```

Every exported function should invoke AFX_MANAGE_STATE as shown in the this sample code:

```
STDAPI DllRegisterServer(void)
{
        AFX_MANAGE_STATE(_afxModuleAddrThis);

        if (!AfxOleRegisterTypeLib(AfxGetInstanceHandle(), _tlid))
                return ResultFromScode(SELFREG_E_TYPELIB);
```

```
        if (!COleObjectFactoryEx::UpdateRegistryAll(TRUE))
                return ResultFromScode(SELFREG_E_CLASS);

        return NOERROR;
}
```

All OLE controls implement a self-registration feature by exporting two functions, DllRegisterServer and DllUnregisterServer. These functions are called directly from outside of the control, so AFX_MANAGE_STATE must be invoked at the beginning of each function.

AFX_MANAGE_STATE should also be used when handling messages sent to a child window created by an OLE control. Such messages are sent directly to the child window, bypassing the call to AFX_MANAGE_STATE in COleControl::WindowProc().

Additional reference words: 1.51 1.00 1.52 1.10 2.00 2.10
KBCategory: kbole kbcode
KBSubcategory: CDKIss

How to Use an OLE Control as an Automation Server

PSS ID Number: Q146120

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SUMMARY
=======

In situations where an OLE container doesn't support control containment, you may want to use an OLE control as an automation server to gain access to its properties and methods. This article explains the necessary modifications you need to make in order for an OLE control to function as a normal automation server.

MORE INFORMATION
================

Prior to Visual C++ 4.0, an OLE Control could be used as an automation server without any modification. However, in MFC 4.0, the framework's implementation of IDispatch::Invoke calls the virtual function IsInvokeAllowed to determine if an automation server is in the appropriate state to handle automation calls. The default implementation in CCmdTarget::IsInvokeAllowed returns TRUE, implying that a server can handle automation calls.

In the case of an OLE control, COleControl::IsInvokeAllowed checks to see if the control has been either initialized or loaded properly through the persistent storage interfaces. If the control has the appropriate state information, then this function returns TRUE. When an OLE control is created as a normal automation server, it is not created as an embedding in the client. Hence, none of the persistent state initialization will take place, which thereby causes IsInvokeAllowed to return FALSE.

In order to use an OLE control only as an automation server, you need to override COleControl::IsInvokeAllowed() and return TRUE. If any of the control's properties and methods should not be accessed when invoked as a normal automation server, then that automation function could be bypassed and/or an error code can be returned when COleControl::m_bInitialized is FALSE.

For example, the default OLE control behavior may pose a problem when an automation client calls CreateDispatch to create the OLE control as an automation server. Prior to making the above modification, CreateDispatch may succeed; however, any automation calls on the OLE control would fail.

Sample Code

```
BOOL CMyOleControl::IsInvokeAllowed (DISPID)
{
```

```
   // You can check to see if COleControl::m_bInitialized is FALSE
   // in your automation functions to limit access.
   return TRUE;
}
```

Additional reference words: kbinf 4.00 ocx OLE Automation error
KBCategory: kbprg kbole kbhowto kbcode
KBSubcategory: MfcOLE CDKIss

How to Use OLE2UI Functionality in 32-bit Applications

PSS ID Number: Q135862

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2

SUMMARY

=======

The OLE2UI library is a 16-bit DLL provided with the 16-bit OLE SDK. It contains a number of APIs that are useful for user interface (UI) related functionality. However, beginning with Windows NT version 3.51, this DLL is no longer provided. OLEDLG.DLL provides the dialog boxes that were included in OLE2UI. Two DLLs (MFCUIW32 and MFCUIA32) essentially wrap the APIs from OLEDLG.DLL, providing the ANSI and Unicode versions of the dialog boxes. If your application uses one of the approximately 122 other APIs that are no longer provided, copy the source code from the 32-bit version provided as a sample in the Visual C++ version 2.x directory ...\SAMPLES\MFC\MFCUIX32.

MORE INFORMATION

================

The two DLLs (MFCUIW32 and MFCUIA32) support the following APIs:

OleUIAddVerbMenu
OleUIBusy
OleUICanConvertOrActivateAs
OleUIChangeIcon
OleUIChangeSource
OleUIConvert
OleUIEditLinks
OleUIInsertObject
OleUIObjectProperties
OleUIPasteSpecial
OleUIPromptUser
OleUIUpdateLinks

OLEDLG.DLL provides the A and W versions of these functions.

To use one of the other functions, you have two options:

- Create a DLL using the sample code, and statically link to the library. This option is best if you were using a lot of the functions.

  -or-

- Include the prototype for the function in one of your header files, and copy the source code from the .cpp file in the MFCUIX32 directory to your .cpp file. This option is probably best if you are using only one or two of the functions. For example, if you would like to use XformRectInPixelsToHimetric(), copy the prototype from \msvc20\... \mfcuix32\olestd.h, line 433, and the source code from \msvc20\ ...

\mfcuix32\oleutl.cpp starting at line 401.

REFERENCES
==========

For a list of the API entry points that have been removed, please see the project definition file (OLE2UI.DEF) in the \MSVC20\SAMPLES\MFC\MFCUIX32 directory.

Additional reference words: kbinf 2.00 2.10 2.20
KBCategory: kbole
KBSubcategory: MFCOle

MFC OLE Classes Do Not Support "Embed in self"

PSS ID Number: Q121949

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:

    - Microsoft Visual C++ for Windows, versions 1.5 and 1.51
    - Microsoft Visual C++, 32-bit Edition, version 2.0, 4.0, 4.1
      on the following platform: x86

SUMMARY
=======

The MFC OLE classes in the versions listed above do not include the ability for a container/server to embed items of its own type within an instance of itself.

Users attempting to embed an OLE item from the container/server into itself will receive the "Failed to create object. Check system registry" dialog box. Further debugging will show that QueryInterface failed to return the IViewObject interface for the embedded item.

MORE INFORMATION
================

While OLE makes "embed in self" possible, it requires the use of the API call OleCreateEmbeddingHelper to connect the OLE2 or OLE32 DLL as the default in-process handler. The MFC OLE classes do not make use of this, and the architecture of an MFC application would make it extremely difficult to integrate this support.

MFC applications rely on handle maps and global tracking of such things as the toolbar and status bar of the application. An OLE item from this same application, if activated for visual editing, would attempt to negotiate space for a second toolbar and status bar within the same frame window. The MFC framework would also attempt to add these objects to its handle map, even though they are already there. Altering this would require substantial unsupported changes to the MFC source code. See article Q120682 for information on how to exclude the current server object from its COleInsertDialog

Additional reference words: kbinf OLE embed 1.00 1.50 1.51 2.00 2.50 2.51 3.00 4.00 4.10
KBCategory: kbole kbprg kbtshoot
KBSubCategory: MfcOLE

OLE Control Container Support in Visual C++ and MFC

PSS ID Number: Q135085

The information in this article applies to:

- Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2

SUMMARY

The Microsoft Foundation Classes included with the versions of Visual C++ listed at the beginning of this article do not provide any support for creating OLE control containers.

MORE INFORMATION

The OLE Control Development Kit (CDK) provides the ability to create OLE controls which can be used in OLE control containers such as Microsoft Visual FoxPro version 3.0 and Microsoft Access version 2.0. However, current versions of MFC and Visual C++ do not provide any support for creating OLE control containers.

The next major version of Visual C++, currently planned for release in the fall of 1995, will provide support for creating OLE control containers.

REFERENCES

Here are sources of information regarding creating an OLE control container:

- "Notes on Implementing an OLE Control Container"

  This article by Kraig Brockschmidt can be found on the Microsoft Developer Network CD (MSDN). It discusses the OLE control container architecture and provides sample code that illustrates how to implement an OLE control container.

- "Implementing OLE Control Containers with MFC and the OLE Control Developer's Kit"

  This article by Mike Blaszczak can be found on the MSDN CD and in the April 1995 issue of the "Microsoft Systems Journal." It discusses different types of OLE control container support and provides sample code for an OLE control container.

- "CONTROLS: Demonstrates How to Test OLE Controls"

  This article can be found on the MSDN CD and in the February 1995 issue of the Microsoft Systems Journal. It includes a sample application that implements a version of the MFC DRAWCLI sample that has been

modified to provide OLE control containment.

These articles and samples provide very useful information regarding creating OLE control containers. However, because the current versions of Visual C++ and MFC do not provide any support for creating OLE control containers, these samples are provided as is, with no additional support.

Microsoft Developer Support does not currently provide any support for the creation or implementation of OLE control containers. Because the next major version of Visual C++ will provide a fully supported OLE control container implementation, Microsoft recommends that developers wait for this release unless they have an immediate need for OLE control containment.

Additional reference words: kbinf 1.50 1.51 1.52 1.52b 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20
KBCategory: kbole
KBSubcategory: CDKIss

OLE Control Containers Must Call AfxEnableControlContainer

PSS ID Number: Q150029

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, Versions 4.0, 4.1

SUMMARY

Using Visual C++ version 4.0, it is possible to construct OLE Control Containers. These containers must call AfxEnableControlContainer() in their CWinApp::InitInstance(). This call is added automatically if OLE Controls support is checked in AppWizard Step 3, but in some cases, this call must be added manually. This article explains why you need to call AfxEnableControlContainer() and the possible problems that might occur if this call is not made.

MORE INFORMATION

AfxEnableControlContainer() is responsible for wiring up the support structure necessary for a CWnd object to contain an OLE Control. This call is added to a project by AppWizard if OLE Controls support is selected. You need to add this call manually in the following cases:

- You are adding control container support to an existing project.

  -or-

- The object that will contain the OLE controls is not created directly by AppWizard, such as an OLE Control.

For example, if you are dynamically creating an OLE Control as a child of another OLE Control, the parent control must call AfxEnableControlContainer() in its constructor.

The following list outlines some of the most common problems (others are possible) that can occur if AfxEnableControlContainer() is not called:

- In many cases, you might receive the following TRACE messages in the Output Window of the debugger:

  ```
  >>> If this dialog has OLE controls:
  >>> AfxEnableControlContainer has not been called yet.
  >>> You should call it in your app's InitInstance function.
  ```

- In other cases, you might encounter the following ASSERT in file Cmdtarg.cpp, line 218:

  ```
  ASSERT(afxOccManager != NULL);
  ```

- If you place a Masked Edit or some other OLE control on a CFormView and then run the application, you might see a message box indicating:

  Failed to create empty document.

Additional reference words: kbinf 4.00 4.10
KBCategory: kbole kbprg kbtshoot
KBSubcategory: CDKIss MfcOLE VCx86

Possible Reasons for OLE Control Registration Failure

PSS ID Number: Q140346

The information in this article applies to:

- Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY

OLE controls can be registered by using Visual C++ from the Tools menu, from the Test Container provided with the Control Development Kit (CDK), or by using the regsvr or regsvr32 applications provided with Visual C++. In some cases, the registration of a control may fail; use this article to help troubleshoot the problem.

MORE INFORMATION

All of the previously mentioned methods for registering an OLE Control use essentially the same technique. LoadLibrary() is called to load the control into memory, GetProcAddress() is called to get the address of the DllRegisterServer() function, and then DllRegisterServer() is called to register the control.

Reasons Why the Registration of a Control May Fail

- One or more of the necessary OLE DLLs is not in the path. Instructions for distributing OLE Controls as well as an explanation of what DLLs are necessary to ship can be found in the Shipctrl.wri file located in the same directory as the CDK.

- The control is loading a DLL other than the OLE DLL, and that DLL is not in the path. When the control is loaded into memory, any DLLs that are implicitly loaded through an import library are also loaded. If any of these DLLs are not in the path, the control is not loaded successfully, so registration fails.

- One or more DLLs may be the wrong version. If the control was built with a newer version of a DLL than the one installed on the computer, the control may not load properly, so registration fails.

- An old version of Ocd25.lib is being linked to. If the control is using the MFC database classes, there may be a problem with the version of the Ocd25.lib file that is being linked to.

- The OLE control is located on a Novell server's remote drive. In this case, the access rights to the .ocx file may be preventing the control from loading. Make sure that the access rights for the .ocx file are set to read-only, shareable access, which is the typical setting for executable files.

Troubleshooting Techniques

If none of the possible causes are true in your case, try the following techniques.

1. With the control project loaded in Visual C++, set the executable for the debug session to the OLE Control Test Container (Tstcon16.exe or Tstcon32.exe). When you start the Test Container (under the debugger), you will get a warning that the Test Container does not contain debug information. Ignore this and proceed.

2. From the Test Container, attempt to register the control. Watch for debug output from the OLE Control DLL or any of its dependent DLLs. If you are running the 16-bit product, remember to run the DBWIN program to receive debug output.

   For information on how to set the executable for a DLL debug session, please see the help topic "Debugging DLLs" in Books Online.

As an alternative, you can attempt to register the control programmatically. First create an MFC AppWizard application selecting Dialog-based application and OLE Automation. Enabling OLE Automation will initialize OLE so that the code to register the control will work properly. In the CWinApp-derived class, you will find the function InitInstance() with the initial code as follows:

```
BOOL CTestregApp::InitInstance()
{
   // Initialize OLE libraries
   if (!AfxOleInit())
   {
      AfxMessageBox(IDP_OLE_INIT_FAILED);
      return FALSE;
   }
```

At this point, add the following code segment, which will allow you to check the return codes from LoadLibrary(), GetProcAddress(), and DllRegisterServer.

```
#ifdef _WIN32
    HINSTANCE hDLL = LoadLibrary("some.ocx");
    if(NULL == hDLL)
    {
        // See Winerror.h for explaination of error code.
        DWORD error = GetLastError();
        TRACE1("LoadLibrary() Failed with: %i\n", error);
        return FALSE;
    }

    typedef HRESULT (CALLBACK *HCRET)(void);
    HCRET lpfnDllRegisterServer;

    lpfnDllRegisterServer =
```

```
            (HCRET)GetProcAddress(hDLL, "DllRegisterServer");
    if(NULL == lpfnDllRegisterServer)
    {
        // See Winerror.h for explaination of error code.
        DWORD error = GetLastError();
        TRACE1("GetProcAddress() Failed with %i\n", error);
        return FALSE;
    }

    if(FAILED((*lpfnDllRegisterServer)()))
    {
        TRACE("DLLRegisterServer() Failed");
        return FALSE;
    }

#else // 16-bit
    HINSTANCE hDLL = LoadLibrary("regtest.ocx");
    if(HINSTANCE_ERROR > hDLL)
    {
        // See LoadLibrary() help for explaination of error code.
        TRACE1("LoadLibrary() Failed with: %i\n", hDLL);
        return FALSE;
    }

    typedef HRESULT (CALLBACK *HCRET)(void);
    HCRET lpfnDllRegisterServer;

    lpfnDllRegisterServer =
            (HCRET)GetProcAddress(hDLL, "DllRegisterServer");
    if(NULL == lpfnDllRegisterServer)
    {
        // See GetProcAddress() help for explaination of error code.
        TRACE("GetProcAddress() Failed");
        return FALSE;
    }

    if(FAILED((*lpfnDllRegisterServer)()))
    {
        TRACE("DLLRegisterServer() Failed");
        return FALSE;
    }
#endif
```

Additional reference words: kbinf 1.51 1.52 1.52b 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.00
KBCategory: kbole kbtshoot kbcode
KBSubcategory: CDKIss

PRB: Adding OnClose Handler to COleControl Class Causes C2660

PSS ID Number: Q152391

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SYMPTOMS

When building an ActiveX control (OLE Control), you may receive the following error message:

```
error C2660: 'OnClose' : function does not take 0 parameters
```

CAUSE

This error occurs when using the Developer Studio or Class Wizard to add a WM_CLOSE message handler to your COleControl derived class. The handler created will have the following format:

```
void CEdt_on_ctrlCtrl::OnClose() {
   // TODO: Add your message handler code here and/or call default
   COleControl::OnClose();
}
```

However, an ActiveX Control is a COM object and, as such, uses a different mechanism than processing a WM_CLOSE message to close down its window. For this reason, COleControl has overridden OnClose with its own version using the following prototype:

```
void OnClose(DWORD dwSaveOption);
```

COleControl::OnClose has one parameter that is the reason for the error message.

RESOLUTION

Since the ActiveX Control doesn't handle the WM_CLOSE message, if you want to do some processing when the OLE control is terminating, you can override the COleControl::OnClose function as follows:

Add the following function prototype to the class definition in the header file for your COleControl derived class (in this example, CEdt_on_ctrlCtr):

```
void OnClose(DWORD dwSaveOption);
```

To replace CEdt_on_ctrlCtrl with the name of your OLE control class, add the following implementation in the .cpp file:

```
void CEdt_on_ctrlCtrl::OnClose(DWORD dwSaveOption) {
```

```
      // TODO: Add your message handler code here and/or call default
      COleControl::OnClose(dwSaveOption);
   }
```

STATUS
======

This behavior is by design.

Additional reference words: VC 2.00 2.10 2.20 4.00 4.10
KBCategory: kbprg kbole kberrmsg
KBSubcategory: MfcOLE CDKIss

PRB: Appearance Property Uses Windows 4.0 WS_EX_CLIENTEDGE

PSS ID Number: Q152000

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SYMPTOMS

MFC Controls support a set of standard properties. One of these properties is Appearance (DISPID_APPEARANCE). Appearance is implemented to give a control a 3-D look. If the control is used on a version of Windows or Windows NT less than 4.0, the Appearance property will not affect the appearance of the control at run time.

CAUSE

At run time, the Appearance property is implemented in an MFC control by setting an extended window style WS_EX_CLIENTEDGE. This window style is not supported on versions of Windows or Windows NT less than 4.0.

RESOLUTION

It is possible to supply your own implementation of the Appearance property although this technique will not work for subclassed Windows controls.

STATUS

This behavior is by design.

MORE INFORMATION

Using the Class Wizard, add the stock Appearance property but choose Get/Set methods rather that accepting the stock implementation. In the control, declare a member variable of type BOOL to represent the Appearance property. The stock Get/Set functions can be copied and changed slightly to manipulate this variable as well as handle the property binding requirements. In the Draw function, draw a 3-D border depending on the value of this variable.

Sample Code

```
   // these get and set functions are the same as the original with a minor
   // change (noted in the code)

   short CTestAppearanceCtrl::GetAppearance()
```

```
{
    return m_MyAppearance;
}

void CTestAppearanceCtrl::SetAppearance(short nNewValue)
{
    if (nNewValue != 0 && nNewValue != 1)
        ThrowError(CTL_E_INVALIDPROPERTYVALUE,
                   AFX_IDP_E_INVALIDPROPERTYVALUE);

    // Is the property changing?
    if (m_MyAppearance == nNewValue)
        return;

    if (!BoundPropertyRequestEdit(DISPID_APPEARANCE))
        SetNotPermitted();

// changed from original
    // do not use the WS_EX_CLIENTEDGE style
    //ASSERT((m_hWnd == NULL) ||
    //  ((GetExStyle() & WS_EX_CLIENTEDGE) == (DWORD)(m_sAppearance ?
    //      WS_EX_CLIENTEDGE : 0)));

    m_MyAppearance = nNewValue;
    SetModifiedFlag();

    //ToggleAppearance(this);
// end of changes

    OnAppearanceChanged();

    BoundPropertyChanged(DISPID_APPEARANCE);
}

void CTestAppearanceCtrl::OnDraw(
            CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    // TODO: Replace the following code with your own drawing code.
    pdc->FillRect(rcBounds,
         CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
    pdc->Ellipse(rcBounds);

    // draw as 3D if m_appearance is set

    if(m_appearance)
    {
        CRect rect(rcBounds);
        pdc->DrawEdge(rect, EDGE_SUNKEN, BF_RECT | BF_ADJUST);
    }
}
```

Additional reference words: 4.00 4.10 DISPID_APPEARANCE
KBCategory: kbole kbprg kbusage kbprb kbtshoot
KBSubcategory: MfcOLE

PRB: Assertion While Switching Property Pages in OLE Control

PSS ID Number: Q140105

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SYMPTOMS

If you have implemented an OLE control that uses a basic MFC CPropertySheet as a user interface, you may receive the following error when trying to switch pages by clicking the tab:

```
Assertion Failed
OC30D.DLL: File DlgCore.CPP Line 194
```

CAUSE

This assertion occurs in _AfxCheckDialogTemplate, when MFC cannot find the dialog template resource for the property page.

RESOLUTION

The solution is to override CPropertyPage::OnSetActive and use AFX_MANAGE_STATE as in the following code:

Sample Code

```
/* Compile options needed - none
   Add the following code to each of your CPropertyPage-derived classes.
*/

BOOL CYourPropPage::OnSetActive()
{

  AFX_MANAGE_STATE(_afxModuleAddrThis);
  return CPropertyPage::OnSetActive();
}
```

The prototype for OnSetActive must also be added to your CPropertyPage-derived class header file.

REFERENCES

For information on AFX_MANAGE_STATE, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q127074
TITLE : How to Use AFX_MANAGE_STATE in an OLE Control

Additional refernce words: kbinf 2.00 2.10 2.20 4.00
KBCategory: kbprg kbui kbole kbprb
KBSubCategory: MfcOLE

PRB: ClassWizard Doesn't Support Property Change Notifications

PSS ID Number: Q146446

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

ClassWizard makes adding event handlers in an OLE control container much easier by providing the required macros, function declarations, and the function body. However, ClassWizard currently doesn't provide support to add handlers for property change requests and property change notifications. You must manually add the macros, function declarations, and the function body required for defining an event sink map entry in order to handle property change notifications. This article explains the steps necessary for handling one such property change notification.

RESOLUTION

The following steps illustrate how to handle property change notifications for the "Caption" stock property of the Circ3 sample control in a dialog based control container application:

1. Place the Circ3 control in the main dialog template of the container application.

2. Add an event sink map manually to the control container as follows:

   a. In the .h file of your Main Dialog class, add a DECLARE_EVENTSINK_MAP() macro inside the AFX_MSG section.

   b. In the .cpp file of your Main Dialog Class, add the following:

```
BEGIN_EVENTSINK_MAP(COleCntrDlg, CDialog)
  //{{AFX_EVENTSINK_MAP(COleCntrDlg)
  //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()
```

      In the this event sink map, COleCntrDlg is the name of your main dialog class.

3. Add the following two functions to the protected section in the .h file of your Main Dialog Class:

```
BOOL CaptionRequest(BOOL *pChange);
BOOL CaptionChanged(void);
```

4. Add the following ON_PROPNOTIFY macro outside of the AFX_EVENTSINK_MAP comments section in the event sink map:

```
   ON_PROPNOTIFY(COleCntrDlg, IDC_CIRC3CTRL1, DISPID_CAPTION,
                  CaptionRequest, CaptionChanged )
```

   In this macro, IDC_CIRC3CTRL1 is the ID of the Circ3 control. Please refer to the ON_PROPNOTIFY documentation for the syntax of the macro and its parameters. The documentation incorrectly states that the changed notification function should take a UINT parameter. However, the property changed notification function takes VOID as its only parameter.

5. Add the following implementations for CaptionRequest and CaptionChanged in the .cpp file of your Main Dialog Class:

```
   // Property Change Notifications
   BOOL COleCntrDlg::CaptionRequest(BOOL *pChange)
   {
       // Perform customized processing here
       AfxMessageBox("IPropertyNotifySink::OnRequestEdit called");

       // Return TRUE to allow the property to change, else return
       // FALSE
       *pChange = TRUE;
       return TRUE;    // Notification handled
   }

   BOOL COleCntrDlg::CaptionChanged(void)
   {
       // Perform customized processing here
       AfxMessageBox("IPropertyNotifySink::OnChanged called");

       return TRUE;    // Notification handled
   }
```

MORE INFORMATION
==========

An OLE control container would typically implement IPropertyNotifySink in order to receive notifications about property changes from an OLE control. The control container creates a sink with this interface and connects it to the control through the connection point mechanism.

The OLE control itself is required to call the methods of IPropertyNotifySink only for those properties marked with the [bindable] and [requestedit] attributes in the object's type information. When the control is about to change a [requestedit] property, it must call IPropertyNotifySink::OnRequestEdit before changing the property and must also honor the action by the sink on return from this call. Also, when the control changes a [bindable] property, it is required to call IPropertyNotifySink::OnChanged. An OLE control will send these notifications only for property changes occurring after it is fully constructed and initialized.

REFERENCES
==========

Online documentation for ON_PROPNOTIFY and ON_PROPNOTIFY_RANGE.
Online documentation for Property Change Notification.

Additional reference words: ocx ole control cdk 4.00
KBCategory: kbprg kbole kbhowto kbprb kbcode
KBSubcategory: MfcOLE CDKIss

PRB: COleControl::Serialize Not Called with VB as Container

PSS ID Number: Q141274

The information in this article applies to:

- Standard and Professional Editions of Microsoft Visual Basic programming system for Windows, version 4.0

- The Microsoft Foundation Classes (MFC) included with:

  - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0
  - Microsoft OLE Control Developer's Kit (CDK)

SYMPTOMS

Although OLE controls are used with certain control containers (such as Microsoft Visual Basic version 4.0), the Serialize function for the control-derived class is not called. These containers allow the control to store its persistent information either by using the property sets or by using the IPersistPropertyBag interface. Both these methods call COleControl::DoPropExchange directly without calling the control's Serialize function.

The resolution section of this article discusses a technique that you can use in the DoPropExchange method to store CObject-derived objects.

CAUSE

COleControl::Serialize is called by the framework when an OLE control container uses one of the following persistent storage interfaces for loading and saving the control: IPersistStorage, IPersistStreamInit, or IPersistMemory.

If a control container uses any other method to store the control's persistent information, then Serialize for the control-derived class will not be called. Microsoft Visual Basic, for example, uses either IPersistPropertyBag or property sets to store the persistent information for an OLE control; therefore, the Serialize function for a control is not called when Visual Basic is used as the control container.

RESOLUTION

Although there is no direct support for serializing CObject-derived objects in COleControl::DoPropExchange, you might want to use the following technique to store objects in an OLE control:

1. Allocate a block of memory with GlobalAlloc.

```
// cbGuess is a guess of how much memory will be needed.
// If more is needed, CSharedFile will reallocate.
```

```
   HGLOBAL hMem = GlobalAlloc(GPTR, cbGuess);
   BYTE *pbMem = (BYTE *)hMem;
```

2. Construct an instance of CSharedFile and attach it to the memory block, starting four bytes in. Because the CSharedFile class is not yet documented, include afxpriv.h:

```
   CSharedFile file;
   file.Attach(pMem + sizeof(DWORD), cbGuess - sizeof(DWORD));
```

3. Construct an instance of CArchive on the file:

```
   CArchive ar(&file, CArchive::store);
```

4. Write the CObject-derived objects into the archive:

```
   // store data in the archive
   // for example, if m_myObject is a CObject-derived object, then
   m_myObject.Serialize(ar);
```

5. Get the length of the file and write it into the first DWORD of the memory block:

```
   *(DWORD*)pbMem = file.GetLength();
```

6. Pass the memory block to PX_Blob:

```
   PX_Blob(pPX, _T("MyObjects"), hMem);
```

This code could be used for loading the objects back out of the blob. For more information about how to use PX_Blob to serialize/de-serialize data, Please see the following article in the Microsoft Knowledge Base:

```
   ARTICLE-ID: Q137333
   TITLE     : DOCERR: How to Use the PX_Blob Function
```

To get optimal performance in IPersistStreamInit, ensure that the OLE control maintains a separate Serialize method that writes the CObjects directly to its archive. Care should be taken to save all of the control's persistent data in both Serialize and DoPropExchange.

STATUS

This behavior is by design.

MORE INFORMATION

An OLE control generated using ControlWizard can read and write its persistent state using one of the following interfaces: IPersistMemory, IPersistStorage, IPersistStreamInit, IPersistPropertyBag (not implemented in versions before Visual C++ 4.x), and IDataObject through the property sets implementation. Each of these interfaces with the exception of IPersistPropertyBag and IDataObject call COleControl::Serialize passing

in a CArchive. This archive could be used to store CObject-derived objects as part of the control's persistence.

Some OLE control containers (like Microsoft Visual Basic) use the "save as text" mechanism in order to allow as much of the OLE control's state to be represented in a human-readable format. For optimizing this mechanism, the interfaces IPropertyBag and IPersistPropertyBag are used and therefore are recommended for containers like Visual Basic. IPropertyBag is implemented by the container and is roughly analogous to IStream. IPersistPropertyBag is implemented by controls and is roughly analogous to IPersistStream(Init).

Visual Basic uses the control's IPersistPropertyBag interface, if one is implemented by the control, or it uses the property sets. Property sets are communicated from and to the control through IDataObject::GetData and IDataObject::SetData, implemented by the control. Note that OLE controls generated using earlier versions of Visual C++ don't provide an implementation for IPersistPropertyBag.

The implementation provided by the MFC framework for IPersistPropertyBag and property sets directly call COleControl::DoPropExchange passing in an instance of either CPropbagPropExchange or CPropsetPropExchange respectively.

REFERENCES
==========

OLE Controls Inside Out - by Adam Denning

Additional reference words: 2.00 2.10 2.20 3.00 3.10 3.20 4.00
KBCategory: kbprg kbole kbprb
KBSubcategory: MfcOLE CDKIss

PRB: COleMessageFilter Doesn't Process WM_PAINT

PSS ID Number: Q152074

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SYMPTOMS
========

If an automation client calls a method of an automation server that brings up a modal dialog, the client area below the dialog may not re-paint itself if the dialog is moved over it.

CAUSE
=====

The painting problem occurs because the WM_PAINT messages are in the queue, but are not dispatched.

RESOLUTION
==========

One solution is to retrieve and dispatch all WM_PAINT messages when IMessageFilter::MessagePending is called. The sample code below demonstrates one way for this resolution to be implemented.

MORE INFORMATION
================

To fix the problem with MFC, create a new class that overrides COleMessageFilter::OnMessagePending() as described below. Then revoke the old message filter and register this new one.

Sample Code

```
   //
   // definition of new COleMessageFilter-derived class
   //
   class CMyMessageFilter : public COleMessageFilter
   {
   public:
      virtual BOOL OnMessagePending(const MSG* pMsg);
   };


   //
   // add a member variable to the CWinApp-derived class
   //
   public:
      CMyMessageFilter MMF;
```

```
   //
   // add to the CWinApp::InitInstance after calling AfxOleInit
   //
      COleMessageFilter* OldFilter = AfxOleGetMessageFilter();
      OldFilter->Revoke();
      MMF.Register();


   //
   // Definition of OnMessageFilter
   //
   CMyMessageFilter::OnMessagePending(const MSG* pMsg)
   {
      MSG msg;

      while (PeekMessage(&msg, NULL, WM_PAINT, WM_PAINT,
                         PM_REMOVE | PM_NOYIELD))
      {
         DispatchMessage(&msg);
      }

      if (pMsg->message == WM_PAINT) return TRUE;

      return FALSE;
   }


   //
   // add to CWinApp::ExitInstance
   //
      MMF.Revoke();
      return CWinApp::ExitInstance();
```

Additional reference words: 2.00 2.10 2.20 4.00 4.10
KBCategory: kbole kbprb kbcode
KBSubcategory: MfcOLE VCx86

PRB: Component Gallery Unable to Import OLE Control

PSS ID Number: Q152392

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SYMPTOMS

Microsoft Developer Studio reports the error message:

   Unable to import OLE control.

Make sure the control contains a valid type library" when attempting to import an ActiveX control (OLE control).

CAUSE

By default, MFC applications use ID binding to call methods in ActiveX controls. Because of this behavior, Developer Studio checks to make sure that the version of the type library is the same as that of the control. If the versions are not the same, Developer Studio reports an error.

This error is usually caused by incrementing the version of the ActiveX Control but not the version of the type library.

For more information on ID binding, please see the following article in the Microsoft Knowledge Base:

   ARTICLE ID: Q138138
   TITLE : Three Binding Types (Late, ID, Early) Possible in VB for Apps

RESOLUTION

Make sure that the version of the ActiveX control matches the version of its type library.

In an MFC control, the version of the control is represented by constants found in the implementation file for the OLE control class:

```
const WORD _wVerMajor = 1;
const WORD _wVerMinor = 0;
```

In the same project, the version for the type library is represented by the version keyword in the .odl file for the OLE control:

```
[ uuid(AFCDB500-BB23-11CF-A685-00AA00A70FC2), version(1.0),
  helpstring("VersionTest OLE Control module"), control ]
```

STATUS
======

This behavior is by design.

MORE INFORMATION
================

An updated version of this control would have version identifiers similar
to the following:

```
   const WORD _wVerMajor = 2;
   const WORD _wVerMinor = 4;

   [ uuid(AFCDB500-BB23-11CF-A685-00AA00A70FC2), version(2.4),
     helpstring("VersionTest OLE Control module"), control ]
```

Additional reference words: 4.00 4.10 import control
KBCategory: kbtool kbole kberrmsg
KBSubcategory: MfcOLE CDKIss VWBIss

PRB: Convert Dialog Doesn't Appear for OLE Object in MS Excel

PSS ID Number: Q135765

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52, 1.52b
  - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SYMPTOMS

When the Edit|<Object>|Convert... menu item is selected for an OLE object that is embedded within a Microsoft Excel document, the Convert dialog box does not appear.

CAUSE

Microsoft Excel displays the Convert dialog for an object only if WriteFmtUserTypeStg has been used to write out a clipboard format and user-readable name for the contents of the object. The MFC libraries do not call this function when creating or saving OLE objects.

RESOLUTION

Call WriteFmtUserTypeStg in the Serialize method of your server's document class.

STATUS

This behavior is by design.

MORE INFORMATION

WriteFmtUserTypeStg should be called as part of your server's implementation of IPersistStorage::InitNew and IPersistStorage::Save. By default, MFC OLE server applications do not call WriteFmtUserTypeStg in their implementation of IPersistStorage::InitNew and IPersistStorage::Save. One simple way to achieve this functionality in an MFC application is to make the call to WriteFmtUserTypeStg in the Serialize method of the server's document class.

Steps to Add a Call to WriteFmtUserTypeStg

1. Provide a private clipboard format for your object.

WriteFmtUserTypeStg serializes a clipboard format and user-readable name for the contents of an object to its storage. The HIERSVR sample shows how to provide a private clipboard format for your object if you do not already have one.

2. Add the DoWriteFmtUserTypeStg helper function to your document class.

   This helper function is used to encapsulate retrieving the clipboard format and user-readable name when calling WriteFmtUserTypeStg. The code in the "Sample Code" section of this article shows how to implement this function for the HIERSVR sample.

3. Modify the server document's Serialize() method and add a call to the DoWriteFmtUserTypeStg helper function, as shown in the following sample code.

Sample Code

```
void CServerDoc::DoWriteFmtUserTypeStg(LPSTORAGE lpStorage)
{
   LPOLEOBJECT lpObject = (LPOLEOBJECT)GetInterface(&IID_IOleObject);
   ASSERT(lpObject != NULL);
   CLSID clsid;
   lpObject->GetUserClassID(&clsid);

   LPTSTR pszUserType = NULL;
   OleRegGetUserType(clsid, USERCLASSTYPE_FULL, &pszUserType);

   if (pszUserType)
   {
      WriteClassStg(lpStorage, clsid);
      WriteFmtUserTypeStg(lpStorage, m_cfPrivate, pszUserType);
      CoTaskMemFree(pszUserType);
   }
}

void CServerDoc::Serialize(CArchive& ar)
{
  ASSERT(m_pRoot != NULL);

  if(IsEmbedded() && ar.IsStoring())
  {
    ASSERT(m_lpRootStg != NULL);
    DoWriteFmtUserTypeStg(m_lpRootStg);
  }

  SerializeFontInfo(ar);
  m_pRoot->Serialize(ar);
}
```

Additional reference words: 1.50 1.51 1.52 1.52b 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.00 4.10
KBCategory: kbole kbprg kbinterop kbcode kbprb
KBSubcategory: MfcOLE

PRB: DoSuperclassPaint Calls WM_PRINT for Windows 95

PSS ID Number: Q147426

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

In some cases, you may need to take a window's custom control and subclass it with an OLE control. However, when the control is built using MFC 4.0 and being used in Windows 95, the OLE control may paint incorrectly.

CAUSE

A typical OnDraw() handler in an OLE control that subclasses a window's custom control makes a call to DoSuperclassPaint(). In Windows 95, The MFC 4.0 implementation of DoSuperClassPaint() has been changed to send a WM_PRINT message to the custom control's window procedure instead of a WM_PAINT message. If the original custom control does not handle WM_PRINT, the OLE control will not paint correctly.

RESOLUTION

It is necessary for the original custom control to handle the WM_PRINTCLIENT message, which is what the DefWindowProc sends in response to receiving a WM_PRINT message. The WM_PRINTCLIENT message should be handled in the same manner as the WM_PAINT message except that the calls to BeginPaint() and EndPaint() should be removed.

STATUS

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbprg kbole kbprb
KBSubcategory: CDKIss MfcOLE

PRB: Drag and Drop with TYMED_FILE Runs Out of File Handles

PSS ID Number: Q135682

The information in this article applies to:

- Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52, 1.52b
- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SYMPTOMS

Calling Open() on the CFile pointer returned from COleDataObject::GetFileData() may return EMFILE. Usually, this will happen after repeated drag and drop operations and subsequent calls to GetFileData(). It is also possible to see memory leaks upon termination of the application, specifically, of memory blocks the size of a CFile object.

CAUSE

The documentation for COleDataObject::GetFileData() is lacking in that is does not mention the fact that the returned CFile pointer is owned by the caller. Therefore, it is the responsibility of the caller to call Close() on or simply delete that pointer. The reason for the EMFILE return code is that the files opened in the drag and drop operation are never closed and therefore the system handles are exhausted. Memory leaks are caused by not deleting the CFile objects that were dynamically allocated by GetFileData().

The documentation for COleDataObject::GetFileData provided with Visual C++ 4.0 has been improved. It mentions the fact that it is the responsibility of the caller to delete the returned CFile object, thereby closing the file.

RESOLUTION

Remember to delete the CFile pointer returned from GetFileData() after using it.

Sample Code

The following OnDrop() function definition illustrates the correct cleanup after using the CFile pointer returned from GetFileData().

```
BOOL CMyView::OnDrop(COleDataObject* pDataObject, DROPEFFECT de,
                        CPoint point)
{
    CFile * pFile = pDataObject->GetFileData(CF_HGLOBAL);
    char lpBuf[100];

    if (NULL != pFile)
```

```
        pFile->Read(lpBuf, 100);
    .
    .
    .
    // Make sure to delete the CFile *
    delete pFile;
    return TRUE;
}
```

Additional reference words: cfile delete stgm_hfile tymed_file 1.50 1.51 1.52 1.52b 2.00 2.10 2.20 4.00
KBCategory: kbole kbprb kbdocerr
KBSubcategory: MfcOLE

PRB: Dynamic Creation of Redistributable Control Fails

PSS ID Number: Q151804

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SYMPTOMS

Attempting to dynamically create an instance of one of the redistributable ActiveX controls supplied with Visual C++ fails on machines where Visual C++ has not been installed. However, creating an instance of the same control on a dialog box succeeds.

CAUSE

The redistributable controls supplied with Visual C++ are licensed.

RESOLUTION

Specify a valid License string when dynamically creating the control.

STATUS

This behavior is by design.

MORE INFORMATION

When Visual C++ is installed on a machine, the entire machine becomes licensed to use the redistributable ActiveX controls as part of the installation process. If an instance of one of the redistributable controls is created dynamically and no License key is specified, the creation will succeed because the control is able to determine that the entire machine is licensed and so will allow itself to be created. However, if the same code that performs the dynamic creation of the control is executed on a machine that has not had Visual C++ installed on it (such as Visual Basic or another application that provides the same control), the creation will fail. In this case the control will not allow itself to be created if a License key is not specified and the machine has not been licensed.

The CWnd::CreateControl method can be used to dynamically create an instance of an ActiveX control. When an ActiveX control is inserted into a Visual C++ project via the Component Gallery, a wrapper class is created that can be used to work with the control. The wrapper class provides two overridden versions of the Create method. One of the versions allows a string to be passed that will be used as the License key when creating the control; the other version does not.

Dynamic creation of one of the redistributable ActiveX controls will fail on a machine that has not had Visual C++ installed on it if any of the following methods are used to create the control:

- The version of the Create method in the wrapper class that does not allow a License key to be specified is used.

- A NULL string is passed to the Create method in the wrapper class that does accept a License key.

- The CWnd::CreateControl method is called with a value of NULL for the License key string.

- IClassFactory::CreateInstanceLic is used with a NULL value as the License key parameter.

Creating instances of the redistributable ActiveX controls via a dialog box template will succeed on a machine that has not had Visual C++ installed. This behavior occurs because the Visual C++ resource editor asks the control for its License key when the dialog template is created. The License key is then cached as a DLGINIT resource in the project's .RC file. At run time, when the MFC dialog box code creates the dialog and its contained controls, the License key is read from the dialog resource and used in the call to IClassFactory2::CreateInstanceLic to create the control.

To allow a licensed control to be dynamically created on a machine that is not licensed requires passing a License key. An application that performs dynamic creation of a control must have the control's License key cached in its own code. This must be done while the application is being designed. For sample code illustrating how to request a License key from an object, refer to the following article in the Microsoft Knowledge Base:

```
  ARTICLE ID: Q151771
  TITLE     : SAMPLE: LicReqst Shows Requesting a License Key from an
              Object
```

The following redistributable controls supplied with Visual C++ 4.1 are licensed and require a License key to be dynamically created on machines that have not had Visual C++ installed:

Control File	Description
anibtn32.ocx	Animated button Control
keysta32.ocx	Key Status Control
mci32.ocx	Multimedia MCI Control
mscomm32.ocx	Communications Control
msmask32.ocx	Masked Edit
grid32.ocx	Grid Control
picclp32.ocx	Picture Clip Control

REFERENCES
==========

For more information, please see:

The Visual C++ documentation on the CWnd::CreateControl method.

The Visual C++ documentation on licensing an OLE control.

The Visual C++ documentation on using controls in a non-dialog container.

The OLE Programmer's Reference documentation on the IClassFactory2 interface.

Additional reference words: 4.00 4.10 ocx cdk
KBCategory: kbole kbprb
KBSubcategory: MfcOLE

PRB: Error C4226 and Defining WIN32

PSS ID Number: Q122306

The information in this article applies to:

- Microsoft Visual C++ 32-bit Edition, version 2.0

SYMPTOMS
========

Compiling a source file that includes WINDOWS.H generates many C4226 errors, such as:

```
C:\MSVC20\INCLUDE\olebase.h (8954) : error C4226: nonstandard extension
 used : '__export' is an obsolete keyword

...

C:\MSVC20\INCLUDE\oleauto.h(293) : error C4226: nonstandard extension
 used : '__huge' is an obsolete keyword

...

C:\MSVC20\INCLUDE\ole2.h(4818) : error C4226: nonstandard extension
 used : '__export' is an obsolete keyword
```

Including these header files individually causes the same problem.

CAUSE
=====

A number of the OLE header files that are included in the WINDOWS.H file require WIN32 to be defined. While _WIN32 is defined for you, WIN32 is not. Because WIN32 is not defined, the header file logic assumes that the definitions are for 16-bit Windows-based applications and uses the __export and __huge keywords.

RESOLUTION
==========

Here are three possible way to define WIN32:

- Define WIN32 before the WINDOWS.H file is included.

  ```
  #define WIN32
  #include <windows.h>
  ```

- Define WIN32 in a compiler switch. From the command line, use /DWIN32. In the development environment, follow these following steps:

  1. Choose Settings from the Projects menu, and click the C/C++ tab.

  2. Select the Preprocessor category.

   3. Add WIN32 to the Preprocessor Definitions field of the dialog box.

 - Add the following lines to the beginning of the WINDOWS.H file:

   ```
   #if( defined (_WIN32) && !defined (WIN32) )
   #define WIN32
   #endif
   ```

   NOTE: This will not take care of the case where individual OLE
   header files are included in an application.

MORE INFORMATION

This problem will most likely occur when building console applications.

NOTE: Although this was by design, this behavior was changed in Microsoft Visual C++, 32-bit Edition, version 4.0.

Additional reference words: 2.00
KBCategory: kbtool kbprg kbole kbprb
KBSubcategory: TlsMisc

PRB: Error Converting Visual C++ OLE Controls from 2.x to 4.0

PSS ID Number: Q139457

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0
- Microsoft OLE Control Developer's Kit (CDK)

SYMPTOMS

When you use the following procedure to convert a project built with the CDK that shipped with Visual C++ 2.x to Visual C++ 4.0, the following error appears:

```
Linking...
LINK : fatal error LNK1104: cannot open file "ocs30d.lib"
Error executing link.exe.
circ1.ocx - 1 error(s), 0 warning(s)
```

Conversion Procedure

1. On the File menu, click Open Workspace.

2. Browse to the project directory.

3. List files of type *.mak.

4. Select the 32-bit project file.

5. Click Rebuild All. After it tries to link, the following error message will appear:

   ```
   Linking...
   LINK : fatal error LNK1104: cannot open file "ocs30d.lib"
   Error executing link.exe.
   circ1.ocx - 1 error(s), 0 warning(s)
   ```

   The actual .lib file that is displayed will be one of four variants of the Ocs30*.lib file depending on how the project is being built.

CAUSE

The reason that the specified file cannot be found is that it no longer exists in Visual C++ version 4.0. Because the Ocs30* libraries were variants of the MFC libraries, the functionality was merged with the Mfc40*.dll. This had several advantages, including better code reuse as one MFC .dll file can be used for both the OLE Control container and the OLE control. The problem is that the ocs30*.lib file is included within the Settings, Link page thereby giving the error Ocs30*.lib cannot be found.

RESOLUTION
==========

Change the settings so that Visual C++ version 4.0 is no longer trying to link in the Ocs30*.lib file. This can be done by following these steps:

1. On the Build menu, click Settings, and then click the Link tab.

2. Click the first of the four project build options under Settings For.

3. Delete Ocs30d.lib from the edit field under Object/Library Modules.

4. Click each of the other three project build options and repeat step 3.

Now when you build your project it will use the Mfc40*.dll as designed.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

Sample Code

```
/* Compile options needed:
*/
```

Try one of the Circ samples that ship with the 2.x CDK.

Additional reference words: visualc ocx ole control ocs30d ocs30 lib dll
KBCategory: kbtool kbenv kbprg kbole kbhowto
KBSubcategory: CDKIss MfcOLE

PRB: Failing to Initialize OLE Generates Out of Memory Error

PSS ID Number: Q128086

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++ for Windows, versions 1.5, 1.51, and 1.52
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 4.0, and 4.1 on the following platform: x86

SYMPTOMS
========

If the OLE .DLL files are not initialized with a call to AfxOleInit(), subsequent OLE calls will fail. For 32-bit MFC applications, debug output sent to a debugging window may show these errors:

```
   Throwing exception of type COleException
   sCode = 0x8007000E E_OUTOFMEMORY

   -or-

   sCode = 0x80030008 STG_E_INSUFFICIENTMEMORY
```

For 16-bit MFC applications, debug output may show:

```
   Throwing exception of type COleException
   sCode = 0x80000002 E_OUTOFMEMORY

   -or-

   sCode = 0x80030008 STG_E_INSUFFICIENTMEMORY
```

RESOLUTION
==========

AfxOleInit() calls OleInitialize(LPMALLOC), which sets the task memory allocator used by OLE. If this is not done, OLE cannot perform memory allocations and any OLE calls will fail. Placing a call to AfxOleInit() in the InitInstance() of an MFC application will fix this problem. The first few lines from the InitInstance() of HIERSVR (MFC OLE sample provided with Visual C++) below, shows the proper syntax for calling AfxOleInit().

Sample Code

```
/* Compile options needed: standard MFC project
*/

BOOL CServerApp::InitInstance()
{
    // OLE 2.0 initialization
```

```
    if (!AfxOleInit())
    {
        AfxMessageBox(IDP_OLE_INIT_FAILED);
        return FALSE;
    }
... function continues ....
```

STATUS
======

This behavior is by design.

Additional reference words: 1.50 1.51 1.52 2.00 2.10 2.50 2.51 2.52 3.00 3.10 4.00 4.10
KBCategory: kbole kbprg kbtshoot kbprb kbcode
KBSubCategory: MfcOLE

PRB: How to Use Automation Error Code & Help from Visual Basic

PSS ID Number: Q122488

The information in this article applies to:

- Standard and Professional Editions of Microsoft Visual Basic programming system for Windows, version 3.0

- The Microsoft Foundation Classes (MFC), included with:

  - Microsoft Visual C++ for Windows, versions 1.0 and 1.5
  - Microsoft Visual C++, 32-bit Edition, version 2.0 on the following platform: x86

SYMPTOMS

Visual Basic version 3.0 does not report automation errors that result from OLE automation exceptions. Instead, it reports an err value of 440 and a description string of "OLE Automation exception." The actual wCode passed by the automation server is lost.

WORKAROUND

The More Information section in this article explains and illustrates a method you can use to use the error information and help context code returned by Visual Basic version 3.0 to start WinHelp from an MFC OLE automation server.

STATUS

This behavior is by design in Visual Basic version 3.0. Visual Basic for Applications, included in Excel version 5.0 and other Microsoft products, returns the wCode correctly.

MORE INFORMATION

When an MFC automation server throws an OLE dispatch exception by calling AfxThrowOleDispatchException, an object of class COleDispatchException is built. Its member variables include an OLE SCODE, a help context, string error description, and wCode (an integer code used by Visual Basic).

The only information that the Visual Basic automation controller preserves is the string description. However, the MFC OLE automation server can preserve the information used in the call to AfxThrowOleDispatchException() and expose an additional "Help" method that uses this information to provide help to the user. From within the Visual Basic error handling code, Err is 440, Error(Err) is "OLE Dispatch Exception" and Error$ is the actual string passed by the server.

Sample Code Snippets

```
/* Compile options needed:

   standard MFC OLE project generated by AppWizard*/
```

1. The method in the Automation server that throws the dispatch exception should retain the help context and wCode as member variable of the automation object as in this function:

```
   void CMyObject::Exception()
   {
      m_nIDContext = <some context>;
      m_nSomeCode = <some code>;
      AfxThrowOleDispatchException(m_nSomeCode,"String",m_nIDContext);
   }
```

2. The Automation server should expose a method that uses those member variables to start WinHelp or another help engine with the actual context from the exception as in this code:

```
   short CMyObject::GetError()
   {
      // From here you can use the member variable code
      // and context to start WinHelp or do whatever
      // help code you need to do ...
      AfxGetApp()->WinHelp(m_nIDContext);
      return m_nSomeCode;
   }
```

3. The Visual Basic error handling routine should call the exposed error method in the Automation server when a dispatch exception occurs. Here is example Visual Basic code:

```
   Sub Command1_Click ()
      On Error GoTo EHandle
      Dim a As object
      Set a = CreateObject("TestAuto")
      ' Next line causes exception:
      a.exception
   EHandle:
      Msg = "The error message for error number "
      Msg = Msg & Err & " is:" & NL & NL
      Msg = Msg & """" & Error(Err) & """"
      MsgBox Msg  ' Display message.
      ' Next line starts WinHelp engine by calling method in server.
      a.GetError
      Exit Sub
   End Sub
```

Additional reference words: 1.00 1.50 2.00 2.50 3.00
KBCategory: kbole kbtshoot kbprg kbinterop kbprb
KBSubcategory: MfcMisc

PRB: LPPICTUREDISP Cannot Be Passed Across Process Boundaries

PSS ID Number: Q150034

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1
    - Microsoft OLE Control Developer's Kit (CDK) versions 1.0, 1.1, 1.2

SYMPTOMS
========

An automation server that implements a Picture object wrapped by the CPictureHolder class in MFC fails when it attempts to pass a pointer to the picture object's IPictureDisp implementation across process boundaries.

CAUSE
=====

IPictureDisp gains access to methods of the Picture object that cannot be marshaled across process boundaries. For example, IPictureDisp supports DISPID_PICT_RENDER to gain access to the Render method of the Picture object. The Render method takes a handle to a device context as the first parameter. Device context handles cannot be marshaled.

Dispatch interfaces can normally be marshalled by using the IDispatch marshaling code, but the Picture object implements IMarshal specifically to cause its marshaling to fail.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

The ClassWizard in Visual C++ allows you to select a return type of LPPICTUREDISP for a method. This method might return the IPictureDisp interface obtained by calling CPictureHolder::GetPictureDispatch. However, if this method is called by a controller running in another process, the method fails and returns an error code of E_FAIL.

Because the Picture object causes the marshaling of IPictureDisp to fail, problems can also occur when trying to obtain an IPictureDisp interface across thread boundaries in a multiple-thread apartment model object.

REFERENCES
==========

For more information on marshaling, please see "Inside OLE," second edition, by Kraig Brockschmidt, Chapter 6, "Local/Remote Transparency,"

published by Microsoft Press.

Additional reference words: 1.50 1.51 1.52 2.00 2.10 2.20 4.00 4.10
KBCategory: kbole kbprg kbbpb
KBSubcategory: MfcOLE CDKIss

PRB: Msmask32.ocx Does Not Work Properly in MFC Containers

PSS ID Number: Q146443

The information in this article applies to:

- The Microsoft Masked Edit Control included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

When the Masked Edit Control's Format property is set, it does not automatically format its contents upon losing focus. When embedded in a Visual Basic 4.0 form or Microsoft Access 7.0 form, the control automatically formats its contents when the control loses focus.

CAUSE

The Masked Edit Control uses a proprietary Visual Basic interface called IVBFormat. The absence of IVBFormat support in a control container prevents the Masked Edit Control from automatically updating its contents as specified by the Format property. Control containers created with Visual C++ do not support IVBFormat. Other control containers, such as Visual FoxPro 3.0 forms, which do not support this interface, will also experience limited functionality from the Masked Edit Control.

RESOLUTION

The IVBFormat interface is proprietary and subject to change. Documentation or specifications for this interface are not available. The Masked Edit Control can be used in a Visual C++ control container application, but the automatic formatting feature will remain disabled. Other properties such as the control's Mask property, will function properly in the absence of IVBFormat support.

STATUS

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbole kbprb
KBSubcategory: CDKIss

PRB: OLE Drop Target Does Not Permit Drop

PSS ID Number: Q139648

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52, 1.52a
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2

SYMPTOMS

When you drag an OLE object over a given drop target window, the cursor feedback indicates that a drop is not allowed. However, the same object can be dropped on other drop targets successfully, and other objects can be dropped on the given drop target.

CAUSE

The object descriptor for the object indicates the size of the object is (0,0), and the drop target window's OnDragOver method is using the object rectangle to determine whether the object is within the client area of the window.

RESOLUTION

If the computed position rectangle for a data object is empty, inflate the rectangle to size (1,1) before testing whether the rectangle is within the client area of the drop target window.

STATUS

This behavior is by design.

MORE INFORMATION

Applications that register a window as an OLE drop target should probably check the position of a data object that is dragged over the window before permitting a drop to occur. For example, the MFC sample program Oclient performs the following check in its OnDragOver method:

```
DROPEFFECT CMainView::OnDragOver(COleDataObject*,
   DWORD grfKeyState, CPoint point)
{
   // adjust target rect by original cursor offset
   point -= m_dragOffset;

   // check for point outside logical area (in hatched region)
```

```
   // GetTotalSize() returns the size passed to SetScrollSizes
   CRect rectScroll(CPoint(0, 0), GetTotalSize());

   CRect rectItem(point,m_dragSize);
   rectItem.OffsetRect(GetDeviceScrollPosition());

   DROPEFFECT de = DROPEFFECT_NONE;
   CRect rectTemp;
   if (rectTemp.IntersectRect(rectScroll, rectItem))
   {
      //... figure out the drop effect
   }

   //...update drop effect

   return de;
}
```

However, for some objects, such as Microsoft Word version 6.0 Documents, the m_dragSize reported by the data object is (0,0). In this case, the call to IntersectRect returns FALSE, and the drop effect returned is DROPEFFECT_NONE.

When the object position rectangle is inflated to size (1,1), IntersectRect returns TRUE and the correct drop effect is returned. In the previous example, you would inflate the rectangle by adding the following lines after the rectItem is declared:

```
   if (rectItem.IsRectEmpty())
   {
      // Some applications might have a null size in the object descriptor
      rectItem.InflateRect(1,1);
   }
```

Additional reference words: 1.50 1.51 1.52 1.52a 2.00 2.10 2.20 2.50 2.51 2.52 2.52a 3.0 3.10 3.2 0Client drag/drop Word
KBCategory: kbole kbprb
KBSubcategory: MfcOLE

PRB: Paste Link Fails in Microsoft Excel

PSS ID Number: Q141440

The information in this article applies to:

 - The Microsoft Foundation Classes (MFC) included with:

    - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0
      on the following platform: x86, Intel, Alpha, MIPS

SYMPTOMS
========

After you attempt to use Paste Link to paste the clipboard contents into Microsoft Excel, Microsoft Excel fails and displays a warning:

   Unable to paste link.

CAUSE
=====

When Microsoft Excel attempts to paste link an object, it looks for a full moniker for the object (OLEWHICHMK_OBJFULL). The full moniker consists of the moniker of the object's container (OLEWHICHMK_CONTAINER) and the moniker of the object relative to its container (OLEWHICHMK_OBJREL). In MFC, if SetItemName() is not used to provide a name for the COleServerItem, the OLEWHICHMK_OBJREL portion of the moniker cannot be provided. This causes Paste Link to fail in Microsoft Excel.

RESOLUTION
==========

Set the name of the item to be paste linked by using the SetItemName() function. The sample code in this article illustrates one way you might do this.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

Steps to Reproduce Behavior

This behavior seems to be unique to Microsoft Excel but may exist in other applications as well.

Run the scribble application (step 7) and save a document. In scribble, on

the Edit menu, click Copy to copy the document object and metafile representation to the clipboard. Run Microsoft Excel, choose Paste Special, and select the Paste Link option. Attempt to use Paste Link to paste the Scrib Document Object or the "Picture." Microsoft Excel will not succeed and will display a warning:

   Unable to paste link.

Sample Code

In the case of scribble, the following code will fix the problem

```
CScribItem::CScribItem(CScribDoc* pContainerDoc)
   : COleServerItem(pContainerDoc, TRUE)
{
    SetItemName("Scribble Item");
}
```

REFERENCES
==========

For more information, see COleServerItem::XOleObject::GetMoniker() in Olesvr2.cpp and IOleObject::GetMoniker() in the OLE 2 Programmer's Reference.

Additional reference words: 2.00 2.50 2.51 2.52 3.00 3.10 3.20 4.0
KBCategory: kbole kbtshoot kbprb kbcode
KBSubcategory: MfcOLE

PRB: Possible Recursion Problems with OLE Controls

PSS ID Number: Q140492

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SYMPTOMS

Attempting to use an OLE control in the same module that implements it may generate an exception caused by a stack overflow. One example of using an OLE control in this manner is placing an instance of an OLE control in that same OLE control's About dialog box.

CAUSE

Calling certain CWnd members can create an infinitely recursive loop, causing a stack overflow to occur.

RESOLUTION

Avoid calling certain CWnd members when using an OLE control in the same module that implements it.

STATUS

This behavior is by design.

MORE INFORMATION

The COleControl class itself has been written to make sure that its members are safe and can be called even when a control is being used in the module that implements it. However, certain CWnd members are not safe to call when a control is being used in this fashion. CWnd members that can cause problems include ShowWindow, MoveWindow, SetWindowPos, EnableWindow, SetWindowText, GetWindowText, and GetWindowTextLength.

There may be other members that can cause the same problem. If a control is being used in the same module that implements it and a stack overflow occurs, the first thing the control programmer should check for is this problem.

Additional reference words: 4.00
KBCategory: kbole kbprb
KBSubcategory: mfcOLE CDKIss

PRB: Potential Limit of 64 Different .OCX Files per Process

PSS ID Number: Q133324

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft OLE Control Developer's Kit (CDK) versions 1.0, 1.1, 1.2

SYMPTOMS

Attempting to create or insert an OLE control may cause the following assertion message to be generated:

```
Assertion Failed: <application>: File ctlmodul.cpp, Line 192
```

CAUSE

The assertion is the function _AfxGetCtlModuleStatics and is the following:

```
ASSERT(_afxCtlTlsIndex != NULL_TLS);
```

The assertion fails because the process creating the OLE control has used up all of its available thread-local storage (TLS) slots.

RESOLUTION

To avoid the problem, do one of the following:

- Package more than one OLE control per .OCX file. A TLS slot is allocated when a process first attaches to an .OCX file. Implementing more than one OLE control per .OCX file will use only one TLS slot per .OCX file while still allowing the process to use each of the OLE controls associated with the .OCX file.

  -or-

- Design the containing application in such a way that it does not need to be able to create so many OLE controls simultaneously.

STATUS

This behavior is by design.

MORE INFORMATION

The 32-bit MFC implementation of OLE controls uses TLS to store control module state information. One TLS slot is allocated every time a process attaches to a new OLE control (.OCX) file. One process may allocate up to

TLS_MINIMUM_AVAILABLE TLS slots. TLS_MINIMUM_AVAILABLE is guaranteed to be at least 64 slots on all systems.

NOTE: TLS_MINIMUM_AVAILABLE is a Win32 limitation rather than an MFC limitation.

This problem will occur when a process tries to access more than TLS_MINIMUM_AVAILABLE TLS slots, whether they are used for control module state information or for some other purpose. TLS_MINIMUM_AVAILABLE is the maximum number of different .OCX files (packages that contain OLE controls) that may be loaded before the assertion occurs. Depending on how your container uses thread local storage, you may encounter the assertion after loading a fewer number of .OCX files.

Note, this does not restrict the number of OLE Controls that can be placed in a container simultaneously. As long as your process uses no more than TLS_MINIMUM_AVAILABLE TLS slots, the number of controls that can be created is constrained only by available memory.

REFERENCES
==========

Chapter 8, Thread-Local Storage, in the book Advanced Windows NT, The Developer's Guide to the Win32 Application Programming Interface by Jeffrey Richter.

Additional reference words: 1.00 1.10 1.20 2.0 2.00 2.1 2.10 2.2 2.20
KBCategory: kbole kbprg kbprb
KBSubcategory: MfcOle

PRB: SetItemRects Causes Server to Change Zoom Factor

PSS ID Number: Q126563

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:

    - Microsoft Visual C++ for Windows, versions 1.50 and 1.51
    - Microsoft Visual C++, 32-bit Edition, versions 2.0 and 2.1

SYMPTOMS
========

Calling COleClientItem::SetItemRects() causes the server to change its zoom-factor.

RESOLUTION
==========

In some cases, you may find it advantageous to have a container change the area provided to an in-place active object without changing the zoom factor of the object. To do this, have the container first set the object's extents by calling COleClientItem::SetExtent and then set the in-place item's window size by calling COleClientItem::SetItemRects.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

COleClientItem::SetItemRects() calls IOleInPlaceObject::SetObjectRects() and passes its parameters. The following is the description of IOleInPlaceObject::SetObjectRects() in Volume 2 of the OLE2 Programmer's Reference:

> The object should compare its width and height with those
> provided by its container (conveyed through lprcPosRect).
> If the comparison does not result in a match, the container
> is applying either scaling or zooming to the object.
> The object must then decide whether it should continue the
> in-place editing in the scale/zoom mode or deactivate."

The object's width and height are determined by the extent information passed to it via COleClientItem::SetExtent() -- which calls IOleObject::SetExtent on the in-place object). Therefore if you want the client application to change the rectangle used by an in-place active object, have it first set the extent of the object in HIMETRIC units to the size of the new rectangle and then set the rectangle.

The following sample code demonstrates this technique. It assumes that

CMyClientItem is derived from COleClientItem and that rect has coordinates specified with respect to the view which contains the object. It returns true if SetItemRects was successful.

Sample Code

```
BOOL CMyClientItem::MySetObjectRects(CRect &rect)
{
  ASSERT_VALID(this);
  ASSERT(m_lpObject != NULL);
  ASSERT(IsInPlaceActive());

  // get the size of this new rectangle
  CSize size = rect.Size();

  // now convert this size to HIMETRIC units
  size.cx = XformHeightInPixelsToHimetric(NULL, size.cx);
  size.cy = XformWidthInPixelsToHimetric(NULL, size.cy);

  // use this size to set the extent of the object
  SetExtent(size);

  // finally set the new size for the in-place window
  return SetItemRects(rect, rect);
}
```

Additional reference words: 1.00 1.50 2.00 2.10 2.50 2.51 3.00 3.10
KBCategory: kbole kbprb kbcode
KBSubcategory: MfcOLE

PRB: Stock Font and Color Property Pages Fail

PSS ID Number: Q137353

The information in this article applies to:

- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2

SYMPTOMS

When you include one of the stock CDK property pages (Font, Color, or Picture) in your application, it may not appear when the property sheet dialog box is displayed. The default and other custom property pages may still be visible.

CAUSE

One possible cause of this problem is the lack of registration of the Oc30.dll file in the registry.

RESOLUTION

Register Oc30.dll using Regsvr32.exe that comes with the CDK and is located in the CDK \Bin directory. Use this line:

```
REGSVR32 OC30.DLL
```

STATUS

Microsoft is researching this behavior and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

Microsoft has seen several cases where Oc30.dll was either not registered at all or only partially registered. We have not been able to isolate the cause of this failure.

Before attempting to register OC30.DLL, confirm that it is in the System32 directory in Windows NT and the System directory in Windows 95.

Steps to Reproduce Behavior

To demonstrate the problem, first confirm that the CIRC3 sample has been compiled and registered. Confirm that the Font and Color property pages are seen when the control is run by looking at the property sheet from the menu selection Edit Properties. Then follow these steps:

1. Delete the Font class ID from the HKEY_CLASSES_ROOT\CLSID. The
   CLSID for the Font property page is:

   {2542F180-3532-1069-A2CD-00AA0034B50B}.

2. Run a container (On the Tools menu, click Test Container in Visual C++),
   and insert the CIRC3 Ole control.

3. On the Edit menu, click Properties, and note that the Font property page
   is no longer visible.

You could also delete the Color property page ID:

   {DDF5A00-B9C0-101A-AF1A-00AA0034B50B}

Or the Picture proper page ID:

   {FC7AF71D-FC74-101A-84ED-08002B2EC713}

NOTE: The class IDs for these property pages are different under Visual C++
version 4.0

Additional reference words: 2.00 2.10 2.20 invisible fail hidden
KBCategory: kbole kbprb kbtshoot
KBSubcategory:

PRB: Unable to Load Control from C:\Path\File.ocx

PSS ID Number: Q149565

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1
  - Microsoft OLE Control Developer's Kit (CDK), version 2.0, 2.1, 2.2

SYMPTOMS

When you try to load an OLE Control generated by using ControlWizard into a Visual Basic 4.0 form in Visual Basic by on the Tools menu, clicking Custom Controls and then clicking the Browse button, the following error message appears:

"Unable to load control from: 'x:\path\file.ocx'"

CAUSE

Visual Basic checks an OLE control to see whether it supports self-registration by searching for the presence of an OLESelfRegister entry in the StringFileInfo section of the control's version information resource. ControlWizard-generated OLE controls do not provide the entry by default.

RESOLUTION

Modify the OLE Control to indicate its support for the self-registration feature. This can be achieved by including the OLESelfRegister entry with an empty value in the StringFileInfo section of the control's version resource. The following steps explain the process in detail.

Step-by-Step Procedure

1. Open the .rc file of the OLE Control as a Text file.

2. Find the StringFileInfo block under Version information.

3. Add the following line to the existing list of VALUE statements:

   ```
   VALUE "OLESelfRegister", "\0"
   ```

4. Build the OLE Control. Now you can load the OLE Control (.ocx file) by clicking Custom Controls on the Tools menu and clicking the Browse.

STATUS

This behavior is by design.

MORE INFORMATION
==========

A COM server typically indicates its support for self-registration by including the OLESelfRegister entry with an empty value in the StringFileInfo of its version resource. This entry is considered optional because the actual self-registration mechanisms do not depend on its presence. This information simply allows a client application to avoid loading a DLL (that is, an in-process server or OLE control) or launching an .exe (that is, a local server) for no gain.

The only requirement for an OCX/DLL to support the self-registration mechanism is to provide the entry points DllRegisterServer and DllUnregisterServer and export them. ControlWizard-generated OLE controls provide these two entry points and export them, though they do not indicate their self-registration support by not including the OLESelfRegister entry by default.

REFERENCES
==========

Please refer to Chapter 5 of Inside OLE, Second Edition, for more information on self-registration.

Additional reference words: 2.00 2.10 2.20 4.00 4.10 load failure
KBCategory: kbole kbhowto kbprb kbcode
KBSubcategory: CDKIss MfcOLE

PRB: VB 4.0 Does Not Support Dual Interfaces in OLE Controls

PSS ID Number: Q151903

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SYMPTOMS

OLE Controls that implement a Dual Interface do not use early binding when placed on a Visual Basic 4.0 form. Even if the Dual Interface is marked as the DEFAULT interface in the OLE Control's ODL file, Visual Basic 4.0 still uses the standard IDispatch interface for all automation calls.

STATUS

This behavior is by design.

MORE INFORMATION

Dual Interface provides an alternative to using the standard IDispatch interface when making OLE Automation calls. This technique is also referred to as Early Binding because type checking is performed at compile time. Dual Interfaces are rapidly becoming popular because they provide increased performance over the standard IDispatch interface.

It is possible to add Dual Interface support to automation servers as well as OLE Controls. The ACDUAL sample provided with Visual C++ 4.1 demonstrates the addition of a Dual Interface to the AutoClick automation server. Tech Note 65, referenced in the References section below, outlines the changes you must make to an automation server to support a Dual Interface. Although ACDUAL and Tech Note 65 refer to automation servers, the information they provide is also applicable to OLE Controls.

Visual Basic 4.0 does support early binding for automation servers that support a Dual Interface, but currently does not support Dual Interface OLE Controls. If you attempt to use the Dual Interface on an OLE Control in Visual Basic 4.0, the standard IDispatch interface is used instead. Future versions of Visual Basic may take advantage of Dual Interface OLE Controls, but Visual Basic 4.0 does not.

REFERENCES

ACDUAL Sample - MFC OLE Samples, Visual C++ 4.1

Tech Note 65: Dual-Interface Support for OLE Automation Servers

Additional reference words: 4.00 4.10

KBCategory: kbole kbinterop kbprb
KBSubcategory: MfcOLE CDKIss

PRB: Wrapper Class for Automation Object May Be Incomplete

PSS ID Number: Q152073

The information in this article applies to:

- The Class Wizard, included with:
  Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SYMPTOMS

An automation object may support a statement and function with the same name even though the statement and function will most likely have different behaviors. Class Wizard will not recognize both the statement and function. However, when generating a new class from the object's type library, the Class Wizard will generate a function and prototype for a single function. This generated function will not give you the ability to call both the function and the statement.

CAUSE

The similarly-named function and statement will share a single entry in the .odl file for the object and a single entry in the type library. The Class Wizard does the best job it can and generates a single function and prototype based on this entry.

RESOLUTION

Given adequate documentation for the statement and function, it is possible to modify the wrapper class by hand so that the statement or function may be called. If the parameter list for the statement and function vary, which may be the case if optional parameters are used, the wrapper function can be overridden. Otherwise, the wrapper for the statement and function will need different names.

STATUS

This behavior is by design.

MORE INFORMATION

An example of this problem is WordBasic's Bold. The Bold statement turns the bold character formatting on and off just as selecting the Bold Toolbar button in Word does. The Bold() function returns a short indicating whether all, some, or none of the characters in a selection are formatted as bold. Visual Basic will call the function if a return value is assigned to the Bold() call. Otherwise, it will call the Bold statement.

The .odl statement for Bold is as follows:

```
   [id(0x00008048), helpstring("Makes the selection bold (toggle)"),
   helpcontext(0x00001d48)] short Bold([in, optional] VARIANT On);
```

The Class Wizard will generate the following function and prototype using this information:

```
   short Bold(const VARIANT& On);

   short WordBasic::Bold(const VARIANT& On)
   {
       short result;
       static BYTE parms[] =
           VTS_VARIANT;
       InvokeHelper(0x8048, DISPATCH_METHOD, VT_I2, (void*)&result, parms,
           &On);
       return result;
   }
```

Using this function will result in the WordBasic Bold() function being called. In order to call the statement, you could define BoldStatement() and BoldFunction() functions. In this case, however, because there is no reason to pass a parameter to the Bold() function, the parameter lists vary and it is possible to overload the wrapper functions as follows:

```
   void Bold(const VARIANT& On);
   short Bold();

   void WordBasic::Bold(const VARIANT& On)
   {
       static BYTE parms[] =
           VTS_VARIANT;
       InvokeHelper(0x8048, DISPATCH_METHOD, VT_EMPTY, NULL, parms,
           &On);
   }

   short WordBasic::Bold()
   {
       short result;
       static BYTE parms[] =
           VTS_VARIANT;
       InvokeHelper(0x8048, DISPATCH_METHOD, VT_I2, (void*)&result, NULL);
       return result;
   }
```

REFERENCES
==========

Microsoft Office Developer's Kit - PART 2 Word Basic Reference, Statements and Functions.

Additional reference words: 2.00 2.10 2.20 4.00 4.10 automation variant
KBCategory: kbprg kbole kbprb
KBSubcategory: MfcOLE

SAMPLE: CFormView-Based Server Shows Metafile When not Active

PSS ID Number: Q143299

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft Visual C++, 32-bit Edition, version 4.0

SUMMARY
=======

The FVServer application demonstrates using a CFormView based application as an OLE server object. One of the problems using CFormView is that the dialog-based view will not render itself into the metafile used to display the application when it has been in-place activated and then deactivated.

This sample demonstrates one method of overcoming this limitation.

Download FVVIEW.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft's World Wide Web Site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and search for FVVIEW.EXE
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib\Mslfiles directory
  Get FVVIEW.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type mssupport
  Double-click the MS Software Library icon
  Find the appropriate product area
  Download FVVIEW.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download FVVIEW.EXE

MORE INFORMATION
================

When you base your server on the CFormView class, you create a dialog template with the resource editor to act as the view of the application. If you activate the application in-place, the drawing occurs in the CFormView::OnDraw(). Usually the CFormView derived class does not implement any drawing.

When the user selects outside the in-place activation area the application goes to the loaded state and any representation on the screen is handled by

rendering a presaved metafile. This metafile is created in the COleServerItem::OnDraw() by passing in a device context (DC) for a metafile and expecting the drawing to occur. In most cases the CView derived class's OnDraw() or a separate function is called to actually render into the metafile. See the Scribble tutorial for an example of how to accomplish this.

In the case of a CFormView-derived class, this technique will not work directly. One method of overcoming this limitation is to draw each of the controls yourself. Some controls may not draw and you may have to use GDI calls such as Rectangle() and LineTo() to actually draw the control.

Another method, which is demonstrated in this sample, is to BitBlt a picture of the screen into a memory device context and save it. Then, when the COleServerItem::OnDraw() is called, BitBlt the saved picture into the metafile DC provided. The side benefit to this method is that you get a Print Preview of the object for free from MFC, because it uses the same metafile.

In the sample, the function CFVView::SetPicture() dynamically creates a memory DC and a bitmap object. The bitmap object must be set to the size of the current view before it is selected into the DC. This is accomplished by setting a member variable in the document class to the size of the view after the view is created. In the sample, the size is computed from the size shown by the resource editor and uses hard-coded values. This seems reasonable as long as you are not dynamically changing the size of the dialog. See the documentation on CFVDoc::OnNewDocument() for the method used to get the size in pixels of a dialog (also see KB article Q125681).

The only problem remaining is determining when the SetPicture() function should be called. There are two locations. The first is in the COleServerDoc derived class's CFVDoc::OnDeactivate(). This function is called twice after the user selects outside the server's in-place activation area. The first call is when the server is going from the UIActive state to the Active state, and the second call is going from the Active state to the Loaded state. We must distinguish between these two calls because the first time it is called, the CFormView is visible and the second time it's called, it is not visible. The server's DoVerb() is called to hide the server between calls to OnDeactivate(). If we simply called SetPicture() during OnDeactivate(), we would BitBlt a blank picture over the good picture in the second call. Simply setting a switch solves that problem.

The second place to call SetPicture() is in CFVView::OnActivateView(), an override of the CView method. Check to make sure the view is being activated, and then call SetPicture(). This last call is necessary for embedding in Microsoft Excel, for example. Microsoft Excel does not deactivate the server such that OnDeactivate() is called.

You must also handle the OnGetExtent() to set the initial size of the CFormView application and handle OnSetExtent() to receive size changes. You should also reset the size when the CFormView::OnSize() is called. If you do not care to react to sizing changes these last calls are not necessary. The sample handles some sizing problems, but not all. Disabling sizing should be straight forward to accomplish.

You must also notify the container whenever the data changes. In this simple application, NotifyChanged() is called in the SetPicture(), so any time the application changes what the user is seeing, call SetPicture(). There may be other places where NotifyChanged() should be called. If you call NotifyChanged() without calling SetPicture(), you will get the last metafile that was saved and it may not show the current state the user sees.

This application is not "bullet-proof", but is instead a starting point for further development.

REFERENCES
==========

"Inside OLE", second edition, by Kraig Brockschmidt.
"OLE 2 Programmer's Reference, Vol 1"

Additional reference words: 4.00 meta-file
KBCategory: kbprg kbfile kbole kbprint kbhowto
KBSubcategory: CodeSam

SAMPLE: CLIBIN.EXE: Converting COleClientItem into CLongBinary

PSS ID Number: Q152533

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SUMMARY
=======

CLIBIN.EXE is a Microsoft Foundation Class sample that demonstrates how to write the data from an OLE Embedded Item to an OLE Object field in an Access 7.0 Database. This particular sample uses the MFC/ODBC classes.

Download CLIBIN.EXE, a self-extracting file, from the following services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile CLIBIN.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get CLIBIN.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and download CLIBIN.EXE

- Microsoft Partner Network (MSPN)
  On MSPN Desktop, double-click the Software Library icon
  Search for CLIBIN.EXE
  Display results and download

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download CLIBIN.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

NOTE: Use the -d option when running CLIBIN.EXE to decompress the file and recreate the proper directory structure.

MORE INFORMATION
================

This sample writes the OLE Embedded Item's data to an Access OLE Object field from within the Container App's OnInsertItem member of the CView class. There is an Access database that goes along with this sample located in the project's directory called "longbin.mdb." Remember to register this .mdb file with the ODBC Administrator.

The following code demonstrates how to accomplish this procedure:

Sample Code

```
    ////////////////////////////////////////////////////////////////
    // Extract COleClientItem data and write it out to an Access Field
    // m_pSet is the recordset object
    // The recordset's CLongBinary member is: m_longbin
    // pItem is the COleClientItem object ( embed item )

    /*************************************************************/
    //NOTE: This is a constant large enough for an average Winword
    // document. Increase this number if the item you will be
    // inserting is larger!!!!!!!!
    const int DOCSIZE = 100000;

    LPPERSISTSTORAGE pPersist = NULL;
    LPLOCKBYTES pLockBytes = NULL;
    LPSTORAGE pStorage = NULL;
    LPVOID pVoid = NULL;
    HRESULT hr = S_OK;

    // make sure the recordset is in Add mode
    m_pSet->AddNew( );

    // free the current memory of the CLongBinary
    GlobalFree( m_pSet->m_longbin.m_hData );

    // pItem ( the COleClientItem object ) has a contained IOleObject
    // interface called m_lpObject.
    // Call QI to get the OLE objects data.
    if( FAILED( pItem->m_lpObject->QueryInterface( IID_IPersistStorage,
                                                   (LPVOID*)&pPersist ) ) )
    {
        AfxMessageBox( _T( "interface not found" ) );
        return;
    }

    // Alloc the CLongBinary
    if( ( m_pSet->m_longbin.m_hData =
                    GlobalAlloc( GMEM_MOVEABLE, DOCSIZE ) ) == NULL )
    {
        AfxMessageBox( _T( "memory error" ) );
        goto term1; // special error condition clean-up label
    }
```

```
    // Now lock it so we can write to it.
    if( ( pVoid = GlobalLock( m_pSet->m_longbin.m_hData ) ) == NULL )
    {
        AfxMessageBox( _T( "memory error" ) );
        goto term1; // special error condition clean-up label
    }

    // Create ILockBytes to be converted to an IStorage
    if( FAILED( CreateILockBytesOnHGlobal( m_pSet->m_longbin.m_hData,
                                           FALSE, &pLockBytes ) ) )
    {
        AfxMessageBox( _T( "interface not found" ) );
        goto term1; // special error condition clean-up label
    }

    // Convert the ILockBytes to an IStorage
    if( FAILED( StgCreateDocfileOnILockBytes( pLockBytes, STGM_DIRECT
               | STGM_CREATE | STGM_READWRITE | STGM_SHARE_EXCLUSIVE,
                                                     0, &pStorage ) ) )
    {
        AfxMessageBox( _T( "interface not found" ) );
        goto term2; // special error condition clean-up label
    }

    // Now we can write the item's persistent data into the storage
    // we've provided.
    if( FAILED( pPersist->Save( pStorage, FALSE ) ) )
    {
        AfxMessageBox( _T( "Save failed" ) );
        goto term3; // special error condition clean-up label
    }

    // manage the field state for CLongBinary RFX
    m_pSet->SetFieldDirty( &m_pSet->m_longbin, TRUE );
    m_pSet->SetFieldNull( &m_pSet->m_longbin, FALSE );

    // set the CLongBinary length
    m_pSet->m_longbin.m_dwDataLength =
                              GlobalSize(m_pSet->m_longbin.m_hData );

    // Update the datasource
    m_pSet->Update( );

    // clean-up
    GlobalFree( m_pSet->m_longbin.m_hData );
    pStorage->Release( );
    pLockBytes->Release( );
    pPersist->Release( );
    return;
```

Additional reference words: 4.00 4.10
KBCategory: kbprg kbfile kbole
KBSubcategory: MfcDatabase dbDao

SAMPLE: DAOPROP.EXE: DaoProp Uses MFC Properties Collection

PSS ID Number: Q152387

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SUMMARY
=======

DAOPROP.EXE demonstrates how to extract information from the Properties collection, which is not supported by the MFC DAO classes. In particular, it demonstrates how to get a list of properties and how to add a property.

In order to properly manage variants, DAOPROP.EXE utilizes #defines and classes defined in the DAO SDK. You will want to install the DAO SDK and note the includes and pragmas in STDAFX.H in order to properly link with the DAO SDK.

Download DAOPROP.EXE, a self-extracting file, from the following services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile DAOPROP.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get DAOPROP.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and download DAOPROP.EXE

- Microsoft Partner Network (MSPN)
  On MSPN Desktop, double-click the Software Library icon
  Search for DAOPROP.EXE
  Display results and download

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download DAOPROP.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591

TITLE : How to Obtain Microsoft Support Files from Online Services

NOTE: Use the -d option when running DAOPROP.EXE to decompress the file and recreate the proper directory structure.

MORE INFORMATION
================

Many DAO objects contain Properties collections but, to date, MFC does not wrap the interface functions to extract properties. The following excerpts from DAOPROP show how to get to the DAO interfaces and obtain properties.

For more information on installing and using the DAO SDK, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q149392
TITLE : How to Use the DAO SDK dbDao Classes with Visual C++ 4.x

Sample Code

The following code demonstrates how to get the list of properties from a given DAO object, and how to create a new property:

```
void CDaoPropDlg::OnButton1()
{
    CDaoDatabase db;
    db.Open( PATH_TO_MDB_FILE );
    CDaoTableDef td( &db );
    td.Open( TABLE );
    DAOProperties* prps;
    DAOProperty* prp;
    COleVariant var;

    TRY
    {
        td.m_pDAOTableDef->get_Properties( &prps );
        int i=0;
        m_ListBox.ResetContent();

        while( prps->get_Item(
                        COleVariant( (short)i++, VT_I2 ), &prp ) == S_OK )
        {
            prp->get_Value( &var );

            if( var.vt == VT_BSTR || var.vt == VT_BSTRT )
            if( strlen( ( LPCTSTR ) var.bstrVal ) > 0 )
            m_ListBox.AddString( ( LPCTSTR )var.bstrVal );
        }
    }
    CATCH ( CDaoException, e )
    {
        AfxMessageBox( e-> m_pErrorInfo->m_strDescription );
    }
    END_CATCH
```

```
    UpdateData(FALSE);
    td.Close();
    db.Close();

}

void CDaoPropDlg::OnButton2()
{
    CDaoDatabase db;
    db.Open( PATH_TO_MDB_FILE );
    CDaoTableDef td( &db );
    td.Open( TABLE );
    DAOProperty* prp;
    DAOProperties* prps;

    UpdateData( TRUE );

    if( m_propname.GetLength() == 0 )
    {
        AfxMessageBox( "Please enter Property Name" );
        return;
    }

    if( m_propval.GetLength() == 0 )
    {
        AfxMessageBox( "Please enter Property Value" );
        return;
    }

    DAO_CHECK(td.m_pDAOTableDef->CreateProperty( STV(  m_propname  ),
                                                 OLTV( dbText ),
                                                 VTV(  m_propval ),
                                                 BTV( 0 ),
                                                 &prp));

    if( FAILED( td.m_pDAOTableDef->get_Properties( &prps ) ) )
    {
        AfxMessageBox( "get_Properties Failed" );
        goto term;
    }

    if( FAILED( prps->Append( prp ) ) )
    {
        AfxMessageBox( "Append Failed" );
        goto term;
    }

    term:
    prp->Release();
    prps->Release();
    td.Close();
    db.Close();
}
```

Additional reference words: 4.00 4.10 prop
KBCategory: kbprg kbole kbfile kbcode
KBSubcategory: MfcDAO MfcDatabase

SAMPLE: DBCTL.EXE: Using ODBC in an ActiveX Control

PSS ID Number: Q152534

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SUMMARY

DBCTL demonstrates a way to use ODBC in an ActiveX Control (OLE Control). A Visual C++ and a Visual Basic client application are included as part of this sample. The Visual C++ client is called DBUSR, the Visual Basic client is called DBVBUSR.

You can find DBCTL.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile DBCTL.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get DBCTL.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and Download DBCTL.EXE

- Microsoft Partner Network (MSPN)
  On MSPN Desktop, double-click the Software Library icon
  Search for DBCTL.EXE
  Display results and download

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download DBCTL.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

NOTE: Use the -d option when running DBCTL.EXE to decompress the file and recreate the proper directory structure.

MORE INFORMATION
================

DBCTL creates and opens a Class Wizard-configured CRecordset object from within the virtual override of COleControl::OnSetClientSite. The CRecordset-derived object is configured to the COURSE table of the STDREG32.MDB that comes with the sample.

The CRecordset-derived object is closed from within its own destructor. The CRecordset object is destructed with a call to the delete operator from within the COleControl derived class destructor.

The control exposes one method, a DisplayRecords method, that simply lists the contents of the recordset in the control.

The OLE Control subclasses a listbox. The string items are added to the listbox by using SendMessage.

The DBUSR.EXE and DBVBUSR.EXE demonstrate using the custom control.

Additional reference words: 4.00 4.10
KBCategory: kbprg kbfile kbole
KBSubcategory: MfcDatabase

SAMPLE: Deriving an OLE Control from a Base Control

PSS ID Number: Q141489

The information in this sample applies to:

   The Microsoft Foundation Classes (MFC) included with:
   Microsoft Visual C++, 32-bit Edition, version 4.0, 4.1

This is the 32-bit version of this sample.

SUMMARY

The SHAPES32 sample illustrates deriving an OLE control from another.

Download SHAPES32.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft's World Wide Web Site on the Internet
  - On the www.microsoft.com home page, click the Support icon
  - In the Microsoft Knowledge Base, search for SHAPES32.EXE
  - Open the article, and click the button to download the file

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type mssupport
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Download SHAPES32.EXE

- Microsoft Download Service (MSDL)
  - Dial (206) 936-6735 to connect to MSDL
  - Download SHAPES32.EXE

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib\Mslfiles directory
  - Get SHAPES32.EXE

MORE INFORMATION

The SHAPES32 sample illustrates deriving an OLE control from a base control, thus allowing the derived control to take advantage of the base control's methods, properties, and events.

The SHAPES32 sample implements a base control class called CBaseShapeCtrl, and two derived control classes called CCircleCtrl and CRectangleCtrl. CCircleCtrl and CRectangleCtrl draw themselves using the properties provided by the base CBaseShapeCtrl class. CCircleCtrl and CRectangleCtrl also allow access to the methods and events implemented in the base CBaseShapeCtrl class.

Properties, Methods, and Events

The CBaseShapeCtrl class provides the base functionality of a simple shape control. It implements the following properties, events, and methods:

Name	Type	Use
FillColor	Property	An OLE_COLOR value which represents the color used to fill the shape.
LineColor	Property	An OLE_COLOR value which represents the color used for the shape's outline.
LineWidth	Property	A short value which represents the line width in pixels of the shape's outline.
BaseMethod1	Method	A test method which, when invoked, fires the BaseEvent1 event. Takes a single parameter of type long.
BaseMethod2	Method	A test method which, when invoked, fires the BaseEvent2 event. Takes a single parameter of type BSTR.
BaseEvent1	Event	A test event. Returns a long.
BaseEvent2	Event	A test event. Returns a BSTR.

The CCircleCtrl class provides the functionality of a simple circle control.

It implements the following properties, events, and methods:

Name	Type	Use
CircleShape	Property	A boolean value. If TRUE the control draws itself as a circle, if FALSE it draws itself as an ellipse.
CircleOffset	Property	A short value which represents the number of pixels offset from the center of the bounding rectangle at which the control will draw itself.
CircleMethod1	Method	A test method which, when invoked, fires the CircleEvent1 event. Takes a single parameter of type long.
CircleMethod2	Method	A test method which, when invoked, fires the CircleEvent2 event. Takes a single parameter of type long.
CircleEvent1	Event	A test event. Returns a long.

CircleEvent2	Event	A test event. Returns a long.

The CRectangleCtrl class provides the functionality of a simple rectangle control. It implements the following properties, events, and methods:

Name	Type	Use
RoundedCorners	Property	A boolean value. If TRUE the control draws itself with rounded corners, if FALSE it draws itself with square corners.
RectangleInset	Property	A short value which represents the number of pixels inside the control's bounding rectangle by which the control will inset itself.
RectMethod1	Method	A test method which, when invoked, fires the RectEvent1 event. Takes no parameters.
RectMethod2	Method	A test method which, when invoked, fires the RectEvent2 event. Takes no parameters.
RectEvent1	Event	A test event. Returns void.
RectEvent2	Event	A test event. Returns void.

Modifying the Base Control Class

Several changes need be made to the code generated by ControlWizard to allow a derived control to cleanly inherit the functionality provided by a base control. The following changes were made to the files generated by ClassWizard:

1. The ON_OLEVERB entry for AFX_IDS_VERB_PROPERTIES in the CBaseShapeCtrl message map was removed:

   ```
   BEGIN_MESSAGE_MAP(CBaseShapeCtrl, COleControl)
     //{{AFX_MSG_MAP(CBaseShapeCtrl)
     //}}AFX_MSG_MAP
     //ON_OLEVERB(AFX_IDS_VERB_PROPERTIES, OnProperties)
   END_MESSAGE_MAP()
   ```

   If this isn't done, control containers will list the Properties... verb twice when the Edit | Control Object menu is selected.

2. The CBaseShapeCtrl::CBaseShapeCtrlFactory::UpdateRegistry function was changed to simply return TRUE:

   ```
   BOOL CBaseShapeCtrl::CBaseShapeCtrlFactory::UpdateRegistry(BOOL
     bRegister)
   {
     return TRUE;
   }
   ```

If this isn't done the base control will get registered and containers will list it along with the derived controls when the user attempts to insert a new control.

3. The CBaseShapePropPage::CBaseShapePropPageFactory::UpdateRegistry function was changed to simply return TRUE:

```
BOOL CBaseShapePropPage::CBaseShapePropPageFactory::UpdateRegistry(
   BOOL bRegister)
 {
   return TRUE;
 }
```

If this isn't done the base class control's property page will get registered and will appear in the registry.

4. The call to InitializeIIDs in the base control class' constructor was removed:

```
CBaseShapeCtrl::CBaseShapeCtrl()
{
  //InitializeIIDs(&IID_DBaseShape, &IID_DBaseShapeEvents);

  // TODO: Initialize your control's instance data here.
}
```

If this isn't done there will be a memory leak upon termination when the cached type information for the base class does not get freed.

5. The base control's section of the project's .ODL file was commented out.

If this isn't done, some containers, for example, Visual Basic 4.0, will fail to load the control.

6. The CBaseShapeCtrl::DoPropExchange method was modified to not call the COleControl::ExchangeVersion function:

```
void CBaseShapeCtrl::DoPropExchange(CPropExchange* pPX)
{
  // ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
  COleControl::DoPropExchange(pPX);

  ...
}
```

The _Version property is serialized via a call to the ExchangeVersion method in the DoPropExchange method of the derived control.

7. The DoPropExchange method of the derived OLE controls were changed to make a call to CBaseShapeCtrl::DoPropExchange instead of COleControl::DoPropExchange:

```
void CCircleCtrl::DoPropExchange(CPropExchange* pPX)
```

```
   {
     ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
     CBaseShapeCtrl::DoPropExchange(pPX);

     ...
   }
```

Manually Updating the Project's ODL File

ClassWizard does not provide any support for developing OLE controls which are derived from another control. As a result, the property, method, and event dispatch ID's used by a derived control class need to be manually updated. MFC Technical Note #39 discusses the dispatch ID numbering scheme used by MFC.

Basically, MFC divides a 32-bit dispatch ID (DISPID) into two parts. The LOWORD of the DISPID contains the distance from the top of the dispatch map (1 relative). The HIWORD contains the distance of the dispatch map from the most derived class (0 relative).

The CCircleCtrl and CRectangleControl classes use this technique with their DISPIDs. For example, the CCircleCtrl derived control inherits the FillColor,LineColor, and LineWidth properties from CBaseShapeCtrl. The DISPID's for these properties need to be manually added to the CCircleCtrl properties section of the project's .ODL file and the HIWORD of the DISPID's needs to be adjusted:

```
//  Primary dispatch interface for CCircleCtrl
[ uuid(A7EC6760-BFED-11CE-8250-524153480001),
  helpstring("Dispatch interface for sample Circle Control"), hidden ]
dispinterface _DCircle
{
  properties:
    // NOTE - ClassWizard will maintain property information here.
    //    Use extreme caution when editing this section.
    //{{AFX_ODL_PROP(CCircleCtrl)
    [id(65537)] OLE_COLOR FillColor;
    [id(65538)] OLE_COLOR LineColor;
    [id(65539)] short LineWidth;
    [id(1)] boolean CircleShape;
    [id(2)] short CircleOffset;
    //}}AFX_ODL_PROP
    ...
```

The FillColor, LineColor, and LineWidth properties implemented in the base class have DISPID's of 1, 2, and 3 respectively. Once the HIWORD portion of these DISPID's has been adjusted they become 65537, 65538, and 65539.

The DISPID's for events are handled differently than for properties and methods. Event DISPID's do not have their HIWORD portion adjusted. Instead, they are sequential. For example, CCircleCtrl inherits the BaseEvent1 and BaseEvent2 events from the base control. The DISPID's for the base events need to be manually added to the CCircleCtrl events section of the project's .ODL file. Also, the DISPID's assigned by ClassWizard for the

events provided by CCircleCtrl (CircleEvent1 and CircleEvent2) need to have their DISPIDs manually updated:

```
[ uuid(A7EC6761-BFED-11CE-8250-524153480001),
  helpstring("Event interface for sample Circle Control") ]
dispinterface _DCircleEvents
{
  properties:
    //  Event interface has no properties

  methods:
    // NOTE - ClassWizard will maintain event information here.
    //    Use extreme caution when editing this section.
    //{{AFX_ODL_EVENT(CCircleCtrl)
    [id(1)] void BaseEvent1(long lParam);
    [id(2)] void BaseEvent2(BSTR pszString);
    [id(3)] void CircleEvent1(long lParam);
    [id(4)] void CircleEvent2(long lParam);
    //}}AFX_ODL_EVENT
};
```

Manually updating the DISPIDs can be problematic because ClassWizard may get confused when it sees the entries for the events in the base class. It's possible that more than one event may be assigned the same DISPID. After adding new events to a derived class, inspect the project's .ODL file and fix any conflicts.

ClassWizard also maintains DISPID values for properties, methods, and events, in an enum member of the COleControl derived class. Again, because ClassWizard does not support deriving an OLE control from another control, it may generate conflicting values for the different DISPIDs. If this happens, manually edit the values so that they are correct and match the values as specified in the .ODL file. Following is the portion of the CCircleCtrl class that shows the DISPIDs used for both the base class and derived class properties, events, and methods:

```
class CCircleCtrl : public CBaseShapeCtrl
{
...

// Dispatch and event IDs
public:
  enum {
  //{{AFX_DISP_ID(CCircleCtrl)
  //dispidFillColor   = 65537L,
  //dispidLineColor   = 65538L,
  //dispidLineWidth   = 65539L,
  //dispidBaseMethod1 = 65540L,
  //dispidBaseMethod2 = 65541L,

  dispidCircleShape   = 1L,
  dispidCircleOffset  = 2L,
  dispidCircleMethod1 = 3L,
  dispidCircleMethod2 = 4L,
```

```
  //eventidBaseEvent1 = 1L,
  //eventidBaseEvent2 = 2L,
  eventidCircleEvent1 = 3L,
  eventidCircleEvent2 = 4L,
  //}}AFX_DISP_ID
};
```

SHAPES is based on a ControlWizard generated OLE control. Files included with the sample which are directly related to deriving an OLE control from a base control are:

SHAPES.ODL -
Shows the modified DISPIDs used by the derived control classes.

BASECTL.CPP -
Provides the implementation of the base CBaseShapeCtrl class.

CIRCCTL.CPP -
Provides the implementation of the derived CCircleCtrl class.

RECTCTL.CPP -
Provides the implementation of the derived CRectangleCtrl class.

REFERENCES

MFC Technical Note #39.

Additional reference words: 4.00 4.10 Shapes Shapes2
KBCategory: kbole kbfile kbcode
KBSubcategory: MfcOLE

SAMPLE: Deriving an OLE Control from a Base Control

PSS ID Number: Q138411

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft OLE Control Developer's Kit (CDK), versions 1.0, 1.1, 1.2

This is the Visual C++ 1.5x and 2.x version of this sample. There is an equivalent Visual C++ 4.x 32-bit sample available under the name SHAPES32.

SUMMARY

The SHAPES sample illustrates how to derive an OLE control from a base control thus allowing the derived control to take advantage of the base control's methods, properties, and events.

Download Shapes2.exe, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type mssupport
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Download Shapes2.exe

- Microsoft Download Service (MSDL)
  - Dial (206) 936-6735 to connect to MSDL
  - Download Shapes2.exe

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib\Mslfiles directory
  - Get Shapes2.exe

MORE INFORMATION

The Shapes2 sample implements a base control class called CBaseShapeCtrl and two derived control classes called CCircleCtrl and CRectangleCtrl. CCircleCtrl and CRectangleCtrl draw themselves using the properties provided by the base CBaseShapeCtrl class. CCircleCtrl and CRectangleCtrl also allow access to the methods and events implemented in the base CBaseShapeCtrl class.

Properties, Methods, and Events

The CBaseShapeCtrl class provides the base functionality of a simple shape control. It implements the following properties, events, and methods:

Name	Type	Use
FillColor	Property	An OLE_COLOR value that represents the color used to fill the shape.
LineColor	Property	An OLE_COLOR value that represents the color used for the shape's outline.
LineWidth	Property	A short value that represents the line width in pixels of the shape's outline.
BaseMethod1	Method	A test method that, when invoked, fires the BaseEvent1 event. Takes a single parameter of type long.
BaseMethod2	Method	A test method that, when invoked, fires the BaseEvent2 event. Takes a single parameter of type BSTR.
BaseEvent1	Event	A test event. Returns a long.
BaseEvent2	Event	A test event. Returns a BSTR.

The CCircleCtrl class provides the functionality of a simple circle control. It implements the following properties, events, and methods:

Name	Type	Use
CircleShape	Property	A boolean value. If TRUE, the control draws itself as a circle. If FALSE, it draws itself as an ellipse.
CircleOffset	Property	A short value that represents the number of pixels offset from the center of the bounding rectangle where the control will draw itself.
CircleMethod1	Method	A test method that, when invoked, fires the CircleEvent1 event. Takes a single parameter of type long.
CircleMethod2	Method	A test method that, when invoked, fires the CircleEvent2 event. Takes a single parameter of type long.
CircleEvent1	Event	A test event. Returns a long.
CircleEvent2	Event	A test event. Returns a long.

The CRectangleCtrl class provides the functionality of a simple rectangle control. It implements the following properties, events, and methods:

Name	Type	Use
RoundedCorners	Property	A boolean value. If TRUE, the control draws itself with rounded corners. If FALSE, it draws itself with square corners.
RectangleInset	Property	A short value that represents the number of pixels inside the control's bounding rectangle where the control will inset itself.
RectMethod1	Method	A test method that, when invoked, fires the RectEvent1 event. Takes no parameters.
RectMethod2	Method	A test method that, when invoked, fires the RectEvent2 event. Takes no parameters.
RectEvent1	Event	A test event. Returns void.
RectEvent2	Event	A test event. Returns void.

Modifying the Base Control Class

Several changes need be made to the code generated by ControlWizard to allow a derived control to cleanly inherit the functionality provided by a base control. The following changes were made to the files generated by ClassWizard:

1. The ON_OLEVERB entry for AFX_IDS_VERB_PROPERTIES in the CBaseShapeCtrl message map was removed:

   ```
   BEGIN_MESSAGE_MAP(CBaseShapeCtrl, COleControl)
     //{{AFX_MSG_MAP(CBaseShapeCtrl)
     //}}AFX_MSG_MAP
     //ON_OLEVERB(AFX_IDS_VERB_PROPERTIES, OnProperties)
   END_MESSAGE_MAP()
   ```

   If this isn't done, control containers will list the Properties verb twice when the the user clicks Control Object on the Edit menu.

2. The CBaseShapeCtrl::CBaseShapeCtrlFactory::UpdateRegistry function was changed to simply return TRUE:

   ```
   BOOL CBaseShapeCtrl::CBaseShapeCtrlFactory::UpdateRegistry(
       BOOL bRegister)
   {
     return TRUE;
   }
   ```

   If this isn't done, the base control is registered and containers will list it along with the derived controls when the user attempts to insert a new control.

3. The CBaseShapePropPage::CBaseShapePropPageFactory::UpdateRegistry function was changed to simply return TRUE:

   ```
   BOOL CBaseShapePropPage::CBaseShapePropPageFactory::UpdateRegistry(
     BOOL bRegister)
   {
     return TRUE;
   }
   ```

   If this isn't done, the base class control's Property page is registered, so it will appear in the registry.

4. The call to InitializeIIDs in the base control class constructor was removed:

   ```
   CBaseShapeCtrl::CBaseShapeCtrl()
   {
     //InitializeIIDs(&IID_DBaseShape, &IID_DBaseShapeEvents);

     // TODO: Initialize your control's instance data here.
   }
   ```

   If this isn't done, there will be a memory leak upon termination
   when the cached type information for the base class is not freed.

5. The code in the base control's section of the project's .odl file was
   changed to comments (commented out). If this isn't done, some
   containers (for example, Visual Basic version 4.0) won't load the
   control.

6. The CBaseShapeCtrl::DoPropExchange method was modified to not call the
   COleControl::ExchangeVersion function:

```
   void CBaseShapeCtrl::DoPropExchange(CPropExchange* pPX)
   {
     // ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
     COleControl::DoPropExchange(pPX);

     ...
   }
```

   If this isn't done, saving the state of the derived OLE control will
   generate an assertion on line 1169 of the Propset.cpp file. The
   assertion is generated because the _Version property has already been
   serialized by a call to the ExchangeVersion method in the DoPropExchange
   method of the derived control.

7. The DoPropExchange method of the derived OLE controls was changed to
   make a call to CBaseShapeCtrl::DoPropExchange instead of
   COleControl::DoPropExchange:

```
   void CCircleCtrl::DoPropExchange(CPropExchange* pPX)
   {
     ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
     CBaseShapeCtrl::DoPropExchange(pPX);

     ...
   }
```

Manually Updating the Project's .Odl File

ClassWizard does not provide any support for developing OLE controls that are derived from another control. As a result, the property, method, and event dispatch ID's used by a derived control class need to be manually updated. MFC Technical Note #39 discusses the dispatch ID numbering scheme used by MFC. Basically, MFC divides a 32-bit dispatch ID (DISPID) into two parts. The LOWORD of the DISPID contains the distance from the top of the dispatch map (1 relative). The HIWORD contains the distance of the dispatch map from the most derived class (0 relative).

The CCircleCtrl and CRectangleControl classes use this technique with their DISPIDs. For example, the CCircleCtrl derived control inherits the FillColor, LineColor, and LineWidth properties from CBaseShapeCtrl. The DISPIDs for these properties need to be manually added to the CCircleCtrl properties section of the project's .odl file and the HIWORD of the DISPIDs needs to be adjusted:

```
//  Primary dispatch interface for CCircleCtrl
[ uuid(A7EC6760-BFED-11CE-8250-524153480001),
  helpstring("Dispatch interface for sample Circle Control"), hidden ]
dispinterface _DCircle
{
  properties:
    // NOTE - ClassWizard will maintain property information here.
    //    Use extreme caution when editing this section.
    //{{AFX_ODL_PROP(CCircleCtrl)
    [id(0x10001)] OLE_COLOR FillColor;
    [id(0x10002)] OLE_COLOR LineColor;
    [id(0x10003)] short LineWidth;
    [id(1)] boolean CircleShape;
    [id(2)] short CircleOffset;
    //}}AFX_ODL_PROP
    ...
```

The FillColor, LineColor, and LineWidth properties implemented in the base class have DISPIDs of 1, 2, and 3 respectively. Once the HIWORD portion of these DISPIDs has been adjusted, they become 0x10001 (65537), 0x10002 (65538), and 0x10003 (65539).

The DISPIDs for events are handled differently from those for properties and methods. Event DISPIDs do not have their HIWORD portion adjusted. Instead, they are sequential. For example, CCircleCtrl inherits the BaseEvent1 and BaseEvent2 events from the base control. The DISPIDs for the base events need to be manually added to the CCircleCtrl events section of the project's .odl file. Also, the DISPIDs assigned by ClassWizard for the events provided by CCircleCtrl (CircleEvent1 and CircleEvent2) need to have their DISPIDs manually updated:

```
[ uuid(A7EC6761-BFED-11CE-8250-524153480001),
  helpstring("Event interface for sample Circle Control") ]
dispinterface _DCircleEvents
{
  properties:
    //  Event interface has no properties

  methods:
    // NOTE - ClassWizard will maintain event information here.
    //    Use extreme caution when editing this section.
    //{{AFX_ODL_EVENT(CCircleCtrl)
    [id(1)] void BaseEvent1(long lParam);
    [id(2)] void BaseEvent2(BSTR pszString);
    [id(3)] void CircleEvent1(long lParam);
    [id(4)] void CircleEvent2(long lParam);
    //}}AFX_ODL_EVENT
};
```

Manually updating the DISPIDs can be problematic because ClassWizard may get confused when it sees the entries for the events in the base class. As a result, more than one event may be assigned the same DISPID. After adding new events to a derived class, inspect the project's .odl file, and fix any conflicts.

ClassWizard also maintains DISPID values for properties, methods, and events, in an enum member of the COleControl derived class. Again, because ClassWizard does not support deriving an OLE control from another control, it may generate conflicting values for the different DISPIDs. If this happens, manually edit the values so that they are correct, and match the values as specified in the .odl file. Following is the portion of the CCircleCtrl class that shows the DISPIDs used for both the base class and derived class properties, events, and methods:

```
class CCircleCtrl : public CBaseShapeCtrl
{
...

// Dispatch and event IDs
public:
  enum {
  //{{AFX_DISP_ID(CCircleCtrl)
  //dispidFillColor   = 65537L,
  //dispidLineColor   = 65538L,
  //dispidLineWidth   = 65539L,
  //dispidBaseMethod1 = 65540L,
  //dispidBaseMethod2 = 65541L,

  dispidCircleShape   = 1L,
  dispidCircleOffset  = 2L,
  dispidCircleMethod1 = 3L,
  dispidCircleMethod2 = 4L,

  //eventidBaseEvent1 = 1L,
  //eventidBaseEvent2 = 2L,
  eventidCircleEvent1 = 3L,
  eventidCircleEvent2 = 4L,
  //}}AFX_DISP_ID
};
```

the Shapes2 sample is based on a ControlWizard-generated OLE control. Files included with the sample that are directly related to deriving an OLE control from a base control are:

   Shapes.odl -
   Shows the modified DISPIDs used by the derived control classes.

   Basectl.cpp -
   Provides the implementation of the base CBaseShapeCtrl class.

   Circctl.cpp -
   Provides the implementation of the derived CCircleCtrl class.

   Rectctl.cpp -
   Provides the implementation of the derived CRectangleCtrl class.

REFERENCES
==========

MFC Technical Note #39.

Additional reference words: 1.50 1.51 1.52 1.52b 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 shapes32
KBCategory: kbole kbfile kbcode
KBSubcategory: CDKIss

SAMPLE: FILEDRAG: How to Support File Drag Server Capabilities

PSS ID Number: Q139067

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SUMMARY

=======

Sample code is available that shows how an application can provide the File Drag drop server capabilities. It shows how to set up the data structures for CF_HDROP and ShellIDList formats. In this sample, the user can enable support for any combination of CF_HDROP and ShellIDList format and see the drag drop result. To obtain this sample:

Download FILEDRAG.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type mssupport
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Download FILEDRAG.EXE

- Microsoft Download Service (MSDL)
  - Dial (206) 936-6735 to connect to MSDL
  - Download FILEDRAG.EXE

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib\Mslfiles directory
  - Get FILEDRAG.EXE

MORE INFORMATION

================

In Windows 3.x, the only application that could act as a File Drag-Drop server was File Manager. Now the file drag-drop server capability is extended to all applications via OLE Drag Drop.

An OLE drag-drop server application that supports the CF_HDROP format can support file drag drop as does File Manager. The Windows system generates the appropriate WM_DROPFILES message for client applications that support File Drop, or the Windows system passes the IDataObject if the client application supports OLE Drop target for CF_HDROP format.

One of the new features of the shell is when you drop files on an icon on the desktop. The Shell opens the particular application with the drop file in it provided the application supports command line arguments where file can be passed as in this example:

```
   notepad.exe myfile.txt
```

This feature could also be provided by a file drag drop server that supports the ShellIDList format, in addition to the CF_HDROP format.

Additional reference words: 4.00
KBCategory: kbole kbfile kbhowto
KBSubcategory:

SAMPLE: How to Use Enumerated Properties in an OLE Control

PSS ID Number: Q137354

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  Microsoft OLE Control Developer's Kit (CDK) versions 1.0, 1.1, 1.2

This is the Visual C++ 1.5x and 2.x version of this sample. There is an equivalent Visual C++ 4.x 32-bit sample available under the name ENPROP32.

SUMMARY

The Enumprop sample illustrates using enumerated properties in an OLE control.

Download Enprop.exe, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download Enprop.exe

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get Enprop.exe

After downloading Enprop.exe, run it in an empty directory using the -d switch (as follows) to automatically create subdirectories to hold the many files it contains:

```
ENPROP.EXE -d
```

MORE INFORMATION

The ENUMPROP sample illustrates how to use enumerated properties in an OLE control. An enumerated property is one that has a specific set of possible values. For example, an OLE control might support a property named FillPattern, which it uses to determine the type of brush to use when drawing the control and limit the values the FillPattern property could be set to. Accepted values for a FillPattern property might be cross-hatched, vertical lines, horizontal lines, and so on.

An OLE control can limit a user's possible choices for a property by presenting a drop list comobo box that lists the possibilities on its property page. However, another mechanism is needed to enable a property browser provided by another application to be able to provide the same type of interface when editing the property.

There are two ways to implement an enumerated property so that a

property browser can gather enough information about the property to provide the right editing interface. One method involves declaring an enum type in the OLE control's .odl file and manually editing the property's declaration to make it be of that type. For example:

```
  ...

  typedef enum
  {
    [helpstring("Solid")]        Solid      = 0,
    [helpstring("Dash")]         Dash       = 1,
    [helpstring("Dot")]          Dot        = 2,
    [helpstring("Dash-Dot")]     DashDot    = 3,
    [helpstring("Dash-Dot-Dot")] DashDotDot = 4,
  } enumLineStyle;

  ...

  properties:
  // NOTE - ClassWizard will maintain property information here.
  //    Use extreme caution when editing this section.
  //{{AFX_ODL_PROP(CEnumCtrl)
    [id(1)] enumLineStyle LineStyle;
  //}}AFX_ODL_PROP

  ...
```

Another method involves overriding the COleControl methods OnGetPredefinedStrings, OnGetPredefinedValue, and OnGetDisplayString.

The ENUMPROP sample illustrates using both methods. The OLE control implemented in the sample supports two custom properties, LineStyle and FillPattern. The LineStyle property is declared as an enum in the control's .odl file. Support for using the FillPattern property as an enumerated property is handled by using overrides of the OnGetPredefinedStrings, OnGetPredefinedValue, and OnGetDisplayString methods.

ENUMPROP is based on a ControlWizard-generated OLE control. Files included with the sample, which are directly related to using enumerated properties, are:

   Enum.odl - Shows declaring the enum type for the LineStyle property.

   Enumctl.cpp - Illustrates overriding the COleControl members
   OnGetPredefinedStrings, OnGetPredefinedValue, and
   OnGetDisplayString.

NOTE: When building the sample, the MKTYPLIB utility will generate the following warning when compiling the .odl file:

```
  warning M0002: Warning near line 35 column 35: specified type is not
  supported by IDispatch::Invoke
```

MKTYPLIB is issuing the warning because the enumLineStyle type is not one of the predefined types supported by IDispatch::Invoke. You can ignore the

warning because the actual value of the LineStyle property is a short, which is a type supported by IDispatch::Invoke.

NOTE: The test container tool provided with the CDK does not implement a drop list comobo box on its property browser page when editing an enumerated property. To see the full benefits of using an enumerated property, use an OLE control container application that provides a property browser with this capability.

RFERENCES
=========

For more information, please see the Per-Property Browsing section in Appendix D, OLE Controls Architecture, in the CDK Books Online.

Additional reference words: 1.50 1.51 1.52 2.00 2.10 2.20 enprop32
KBCategory: kbole kbcode kbfile
KBSubcategory: CDKIss VCx86

SAMPLE: How to Use Enumerated Properties in an OLE Control

PSS ID Number: Q141488

The information in this sample applies to:

   The Microsoft Foundation Classes (MFC), included with:
   Microsoft Visual C++, 32-bit Edition, version 4.0, 4.1

This is the 32-bit version of this sample.

SUMMARY

The ENPROP32 sample illustrates using enumerated properties in an OLE control.

Download ENPROP32.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type mssupport
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Download ENPROP32.EXE

- Microsoft Download Service (MSDL)
  - Dial (206) 936-6735 to connect to MSDL
  - Download ENPROP32.EXE

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib\Mslfiles directory
  - Get ENPROP32.EXE

MORE INFORMATION

The ENPROP32 sample illustrates using enumerated properties in an OLE control. An enumerated property is one which has a specific set of possible values. For example, an OLE control might support a property named FillPattern which it uses to determine the type of brush to use when drawing the control. The control would want to limit the values the FillPattern property could be set to. Accepted values for a FillPattern property might be cross-hatched, vertical lines, horizontal lines, etc.

An OLE control can limit the values a user has to choose from for a property by using a Drop List combo box on its property page. However, another mechanism is needed to enable a property browser provided by another application to be able to provide the same type of interface when editing the property.

There are two ways to implement an enumerated property so that a property browser can gather enough information about the property to provide the right editing interface. One method involves declaring an enum type in the OLE control's .odl file and manually editing the property's declaration to make it be of that type. For example:

```
  ...

  typedef enum
  {
    [helpstring("Solid")]        Solid      = 0,
    [helpstring("Dash")]         Dash       = 1,
    [helpstring("Dot")]          Dot        = 2,
    [helpstring("Dash-Dot")]     DashDot    = 3,
    [helpstring("Dash-Dot-Dot")] DashDotDot = 4,
  } enumLineStyle;

  ...

  properties:
  // NOTE - ClassWizard will maintain property information here.
  //    Use extreme caution when editing this section.
  //{{AFX_ODL_PROP(CEnumCtrl)
    [id(1)] enumLineStyle LineStyle;
  //}}AFX_ODL_PROP
```

Another method involves overriding the COleControl methods OnGetPredefinedStrings, OnGetPredefinedValue, and OnGetDisplayString. The ENUMPROP sample illustrates using both methods. The OLE control implemented in the sample supports two custom properties, LineStyle and FillPattern. The LineStyle property is declared as an enum in the control's .odl file. Support for using the FillPattern property as an enumerated property is handled via overrides of the OnGetPredefinedStrings, OnGetPredefinedValue, and OnGetDisplayString methods.

ENPROP32 is based on a ControlWizard generated OLE control. Files included with the sample, which are directly related to using enumerated properties, are:

   ENUM.ODL -
   Shows declaring the enum type for the LineStyle property.

   ENUMCTL.CPP -
   Illustrates overriding the COleControl members
   OnGetPredefinedStrings, OnGetPredefinedValue, and
   OnGetDisplayString.

RFERENCES
=========

The Per-Property Browsing section in Appendix D, OLE Controls Architecture, in the CDK Books Online.

NOTE: The OLE Control Test Container provided with Visual C++ does not

implement a Drop List combobox on its property browser page when editing an enumerated property. To see the full benefits of using an enumerated property, use an OLE control container application which provides a property browser with this capability, such as Visual Basic 4.0.

Additional reference words: 4.00 4.10 Enumprop
KBCategory: kbole kbcode kbfile
KBSubcategory: MfcOLE

SAMPLE: Insert OLE Object Capabilities to the RichEdit Control

PSS ID Number: Q141549

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY

RichEdit control supports embedding and inplace-activating OLE Objects. To add these capabilities to a RichEdit control in your application, an independent service vendor (ISV) must implement certain OLE interfaces. The RichEdit sample shows how these interfaces are implemented and other implementation details that are required in conjunction with the OLE interfaces to embed and inplace-activate OLE objects.

MORE INFORMATION

An ISV must implement the following OLE interfaces to add the embedding and inplace-activating capabilities to the RichEdit control in their application:

- IRichEditOleCallback, which is used by a RichEdit control to retrieve OLE-related information from its client. This interface is passed by the client application to the RichEdit control via EM_SETOLEINTERFACE message. This is a RichEdit control's custom interface.

- IOleInPlaceFrame, which is used to control the container's top-level frame window. In addition, implementing this interface involves implementing IOleWindow and IOleInPlaceUIWindow interfaces also, because IOleInPlaceFrame is inherited from IOleInPlaceUIWindow, which in turn inherits IOleWindow.

There are a few caveats that are important to enable this feature work to properly:

- The Frame window, which will be the parent of the tool bar and status bar, needs to be created with the WS_CLIPCHILDREN style. If this style is not present, your applications will exhibit some painting problems when an object is inplace-active in your RichEdit control.

- The RichEdit control itself should be created with WS_CLIPSIBLING style. Here too, if the style is not present, RichEdit control will exhibit painting problems when the Object creates child windows during inplace-active.

- When destroying the RichEdit control, your application should deactivate

any inplace-active Object and call IOleObject->Close() on all the embedded objects in the RichEdit control. If this is not done, some object applications may not close down, thus causing them to stay in memory, even after the RichEdit control is destroyed. Here is a code snippet that demonstrates how to handle closing of OLE Objects:

```
   if (m_pRichEditOle)
   {
      HRESULT hr = 0;

      //
      // Start by getting the total number of objects in the control.
      //
      int objectCount = m_pRichEditOle->GetObjectCount();

      //
      // Loop through each object in the control and if active
      // deactivate, and if open, close.
      //
      for (int i = 0; i < objectCount; i++)
      {
         REOBJECT reObj;
          ZeroMemory(&reObj, sizeof(REOBJECT));
         reObj.cbStruct  = sizeof(REOBJECT);

         //
         // Get the Nth object
         //
         hr = m_pRichEditOle->GetObject(i,&reObj,REO_GETOBJ_POLEOBJ);
         if(SUCCEEDED(hr))
         {
            //
            // If active, deactivate.
            //
            if (reObj.dwFlags && REO_INPLACEACTIVE)
               m_pRichEditOle->InPlaceDeactivate();

            //
            // If the object is open, close it.
            //
            if(reObj.dwFlags&&REO_OPEN)
               hr = reObj.poleobj->Close(OLECLOSE_NOSAVE);

               reObj.poleobj->Release();
         }
      }
      m_pRichEditOle->Release();
   }
```

Download Richedit.exe, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type mssupport

Double-click the MS Software Library icon
Find the appropriate product area
Download Richedit.exe

- Microsoft Download Service (MSDL)
Dial (206) 936-6735 to connect to MSDL
Download Richedit.exe

- Internet (anonymous FTP)
ftp ftp.microsoft.com
Change to the Softlib\Mslfiles directory
Get Richedit.exe

Additional reference words: 4.00 1.30 softlib
KBCategory: kbui kbole kbhowto kbfile
KBSubcategory:

SAMPLE:CONDOC.EXE:Extract Container/Document Info with MFC DAO

PSS ID Number: Q152316

The information in this article applies to:

- The Microsoft Foundation Classes (MFC), included with:
  Microsoft Visual C++, 32-bit Edition, versions 4.0, 4.1

SUMMARY

=======

MFC does not currently provide methods for extracting container and document information from DAO collections. The sample, CONDOC.EXE demonstrates how to implement this extraction using MFC DAO objects and, in particular, demonstrates using DAO OLE interfaces not directly available from the MFC DAO classes.

Containers provide an application-independent way for an application to store objects in a database. For example, Microsoft Access uses separate containers to store forms, reports, macros, and modules. This is in a format native only to Access. However, the code below shows how to see which containers are stored in an Access database.

Documents are used to store a specific instance of an application-specific object within a container, such as a form, a report, etc. A container can have more than one document collection. To repeat, Microsoft Access documents and containers are in a format native to Access and not available programatically from other products except through OLE Automation using Access itself.

Download CONDOC.EXE, a self-extracting file, from the following services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile CONDOC.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get CONDOC.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and download CONDOC.EXE

- Microsoft Partner Network (MSPN)
  On MSPN Desktop, double-click the Software Library icon
  Search for CONDOC.EXE

```
      Display results and download

 - Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download CONDOC.EXE

For additional information about downloading, please see the following
article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q119591
   TITLE     : How to Obtain Microsoft Support Files from Online Services

NOTE: Use the -d option when running CONDOC.EXE to decompress the file and
recreate the proper directory structure.

MORE INFORMATION
================

Sample Code
-----------

The following code demonstrates how to extract Container and Document
information and store it in a Tree View control:

   // Command handler which performs the pertinent operations
   ////////////////////////////////////////////////////////////////////
   //
   void CCONDOCView::OnGetinfoOpenMDB()
   {
       // Prepare the open file dialog
       static char BASED_CODE szFilter[] = _
                            T( "Access Database Files  (*.mdb) | *.mdb" );

       CFileDialog dlg(TRUE, "mdb", NULL,
                       OFN_HIDEREADONLY| OFN_OVERWRITEPROMPT, szFilter);

       // To be retrieved by the open file dialog
       CString strFile;

       // open file dialog
       if(dlg.DoModal() == IDOK)
       {
           strFile = dlg.m_ofn.lpstrFile;

           if(strFile.Right(3) != "mdb") // verify extension
           {
               AfxMessageBox("File Extension error");
               return;
           }
       }

       // you will display the info in a tree control
       CTreeCtrl &tree = GetTreeCtrl();

       // the root of all items!
```

```
HTREEITEM itemDatabase =    tree.InsertItem(
                           dlg.m_ofn.lpstrFile );

// An MFC DAO object
CDaoDatabase db;

// straight DAO interface pointer variables
ConDoctainers *pContainers=NULL;
ConDoctainer* pCon=NULL;
DAODocuments* pDocuments=NULL;
DAODocument* pDoc=NULL;

// To be used for iterating thru the collections
short count;
short doccount;

// open the .MDB
try
{
    db.Open( strFile );
}
catch( CDaoException* e )
{
    AfxMessageBox( e->m_pErrorInfo->m_strDescription );
    return;
}

// Using the DAODatabase interface pointer method contained
// by the MFC/DAO class CDaoDatabase, you retrieve the interface's
// set of collections.
if( FAILED( db.m_pDAODatabase->get_Containers( &pContainers ) ) )
{
    AfxMessageBox( "Collection not found" );
    db.Close();
    return;
}

// Once you have the set of collections, you iterate thru each
// individual collection, display some of its attributes,
// then get the document collection on the container,
// and iterate thru the individual documents, displaying some
// of the attributes of the documents.
// See the dbdao.h in the DAOSDK\INCLUDE for more information

try
{
    // get the count property of the containers collection
    pContainers->get_Count( &count );

    HTREEITEM itemContainers = tree.InsertItem( _T( "Containers" )
                                                , itemDatabase );

    // iterate thru the containers
    while( --count > 0 )
    {
```

```
// get an individual container and iterate thru its
//attributes
pContainers->get_Item( COleVariant( count, VT_I2 ),
                                            &pCon );
BSTR bstr=NULL;

pCon->get_Name( &bstr );
CString strName( ( LPCTSTR ) bstr );
HTREEITEM itemName = tree.InsertItem( strName,
                                   itemContainers );

pCon->get_Owner( &bstr );
CString strOwner(_T("Owner: ") );
strOwner += (LPCTSTR) bstr;
HTREEITEM itemOwner = tree.InsertItem( strOwner,
                                      itemName );

pCon->get_UserName( &bstr );
CString strUserName(_T("User Name: ") );
strUserName += (LPCTSTR) bstr;
HTREEITEM itemUserName = tree.InsertItem( strUserName,
                                           itemName );

// Now get the documents collection
if( FAILED ( pCon->get_Documents( &pDocuments ) ) )
{
    AfxMessageBox( "Collection not found" );
    db.Close();
    return;
}

// Now get the number of documents contained in the
// container
pDocuments->get_Count( &doccount );
HTREEITEM itemDocuments = tree.InsertItem(
                             _T( "Documents"), itemName );

// iterate thru the documents
while( --doccount > 0 )
{
    // get an individual document and iterate thru
    // some of its attributes.
    pDocuments->get_Item( COleVariant( doccount, VT_I2 ),
                                                 &pDoc );

    pDoc->get_Name( &bstr );
    CString strDocName( ( LPCTSTR ) bstr );
    HTREEITEM itemDocName = tree.InsertItem( strDocName,
                                        itemDocuments );

    pDoc->get_Owner( &bstr );
    CString strDocOwner(_T("Owner: ") );
    strDocOwner += (LPCTSTR) bstr;
    tree.InsertItem( strDocOwner, itemDocName );
```

```
                    pCon->get_UserName( &bstr );
                    CString strDocUserName(_T("User Name: ") );
                    strDocUserName += (LPCTSTR) bstr;
                    tree.InsertItem( strDocUserName, itemDocName );
                }
            }
        }
        catch( CDaoException*  e)
        {
            AfxMessageBox( e->m_pErrorInfo->m_strDescription );
            db.Close();
            return;
        }
        db.Close();
    }
    ////////////////////////////////////////////////////////////////////
    // END OF SAMPLE CODE
```

REFERENCES
==========

Microsoft Jet Database Engine Programmer's Guide, ISBN #: 1-55615-877-7.

Additional reference words: 4.00 4.10 prop
KBCategory: kbprg kbole kbfile kbcode
KBSubcategory: MfcDAO MfcDatabase

Tutorial: Debugging OLE Applications

PSS ID Number: Q122680

The information in this article applies to:

- The Visual Workbench Integrated Debugger, included with: Microsoft Visual C++, 32-bit Edition, version 2.0 on the following platform: x86

SUMMARY

This article provides a tutorial to assist you in learning how to use the Visual Workbench Integrated Debugger.

The Visual C++ integrated debugger supports simultaneous debugging of OLE client and server applications. You can seamlessly step across and into OLE clients and servers, with the ability to step across OLE Remote Procedure Calls. A second instance of the debugger is automatically spawned the first time an OLE client calls into an OLE server.

When building OLE servers, you may want to debug them in the context of being activated by an OLE container, thus debugging both the container and the server at the same time. This tutorial provides an example of how to debug an OLE server when the main debuggee is an OLE container. It shows how a second instance of the debugger is automatically spawned when an OLE client calls into an OLE server. The tutorial is designed to lead you step by step through the code that creates the OLE server object and establishes the connection between the OLE container and the OLE server. This is useful in tracking down problems that occur when the OLE server does not get created or initialized correctly.

All regular debugging facilities are available as you debug your OLE application.

MORE INFORMATION

Preparing to Debug

To prepare for OLE Client/Server debugging:

1. Open the project for the OLE application and build a version with symbolic debugging information.

2. Choose Options from the Tools menu.

3. Select the Debug tab.

4. Ensure that the OLE RPC Debugging check box and the Just-In-Time (JIT) Debugging boxes are checked. (You must have Windows NT administrator privileges to enable OLE RPC Debugging.)

5. Choose OK. The information is now stored with your project.

6. Set breakpoints at the points in the source files for your OLE application where you want to determine the state of the application.

7. From the Debug menu, choose Go or press the F5 key to start the debugger.

Viewing Derived-Most Types

The QuickWatch dialog box provides support for the automatic downcast of pointers in OLE and MFC debugging. The debugger automatically detects when a pointer is pointing to a subclass of the type it is required to point to. When the pointer is expanded, Visual C++ will add an extra member, which looks like another base class and indicates the derived-most type. For example, if you are displaying a pointer to a CObject and the pointer really points to a CComboBox, the QuickWatch expression evaluator will recognize this and introduce a pseudo CComboBox member so you can access the CComboBox members.

The rest of the tutorial takes you through a debug session, using MFC sample code.

Creating the Object

1. Build debug versions of the Microsoft Foundation Classes (MFC) samples in the development environment:

    - MSVC20\SAMPLES\MFC\CONTAIN\STEP2

    - MSVC20\SAMPLES\MFC\SCRIBBLE\STEP7

2. Run the SCRIBBLE.EXE file, built in step 1, to update the registry to point to this executable file.

3. Load the CONTAIN\STEP2 project into the Visual Workbench.

4. Choose Options from the Tools menu, select the Debug tab, and make sure OLE RPC Debugging is enabled, as described in step 4 of the "Preparing to Debug" section in this article.

5. Open the CONTRVW.CPP file in CONTAIN\STEP2, and set a breakpoint on line 139, which contains a call to CreateItem.

6. Press the F5 key to run CONTAIN.EXE. In the CONTAIN main menu, choose Insert New Object from the Edit menu.

7. From the resulting Object Type list, choose Scrib Document, and then choose the OK button. At this point, the debugger should stop at the breakpoint you set in step 5. This is the call to the Insert Object dialog's CreateItem function. The purpose of CreateItem is to create and initialize an object of the type you selected from the dialog box. The

CreateItem function is passed a CCntrItem object and it uses the object to handle this process.

For a brief overview of what CCntrItem does, see its class definition in CNTRITEM.H. Then look at the definition of COleClientItem (which CCntrItem is derived from) on line 399 in AFXOLE.H.

8. Press the F8 key to step into the call to CreateItem. You are in the MFC source file OLEDLGS1.CPP. Press the F10 key to step over instructions until you get to the call to pNewItem->CreateItem on line 104.

9. Press the F8 key to step into the call to pNewItem->CreateItem. Then press the F10 key to step over instructions until you get to line 594, which contains a call to OleCreate. While stepping through the code, read the comments and observe that storage is allocated for the object and its rendering format is established.

10. Step into the call to OleCreate. At this point, execution proceeds through the RPC mechanism to the server code itself. Therefore, as the server code begins to execute, a new instance of the debugger is created in which to debug the server. A pseudo project for SCRIBBLE.EXE is loaded (as in JIT debugging), and the instruction pointer is set at the call to COleServerDoc::XPersistStorage::InitNew in OLESRV1.CPP. If you installed the .DBG files (see the "NT System Symbols Setup" icon in your Visual C++ program group), your callstack will include fully decorated names.

11. Step over instructions in Scribble's InitNew until you reach the call to pThis->OnNewEmbedding on line 1707. Then step into OnNewEmbedding.

12. Step over lines until you reach the call to OnNewDocument on line 856. Then step into the call to COleServerDoc::OnNewDocument. You are inside COleLinkingDoc::OnNewDocument. (COleServerDoc is derived from COleLinkingDoc.) Notice the code in this small function. It creates a new document object and attaches it to the server (Scribble).

13. Press SHIFT+F7 twice to step out twice to get back into COleServerDoc::XPersistStorage::InitNew, where you first came into Scribble.

14. Step out one more time. This will cause the container to return from its call to OleCreate, the function that took you into Scribble in the first place. At this time, the instance of the debugger that has Contain loaded gets the focus and you are back in OLECLI1.CPP immediately following the call to OleCreate. The embedded Scribble object has now been created, but it is not yet fully initialized.

15. Step into the next line, the call to FinishCreate. Step through the FinishCreate code to see how OLE finalizes the connection between the container and the server, then step out of FinishCreate.

Now a Scribble object has been created and initialized in memory, but it is not yet editable in the container; Scribble hasn't been activated. In fact, Contain still has only an IUnknown interface to Scribble. You can see this by expanding the lpClientSite variable in the Locals Window.

Activating the Object

1. Step out two more times to get back to Contain's OnInsertObject function in CONTRVW.CPP.

2. In Contain's OnInsertObject function, step over five times to get to the call to DoVerb on line 149. This function activates Scribble.

3. Step into the call to DoVerb. Then step over a few lines until you come to the call to Activate on line 71.

4. Step into Activate. Now you are in OLECLI3.CPP. Before going on, scan through the code for Activate. Notice that a rectangle is first created for the embedded Scribble item to live in. Then GetClientSite is called to establish an interface back to the container for the Scribble server. Then the server's DoVerb function is called to pass both of these to the server.

5. Step to line 75. Then step into the call to DoVerb. At this point, execution proceeds once again through the RPC mechanism to the server code itself. As you would expect, the instruction pointer is pointing to the first instruction in the server's COleServerDoc:: XOleObject::DoVerb function.

6. Step to line 2064, the call to OnDoVerb. Then step into OnDoVerb. OnDoVerb consists of a switch statement that executes the command (verb) passed to it. In this case, the command is OLEIVERB_SHOW. Step over instructions until you get to OnShow.

7. Step into OnShow. Then step three lines to ActivateInPlace, and step into ActivateInPlace.

   The ActivateInPlace function does many things, and is worth examining in detail. While it is beyond the scope of this tutorial to go into all the details, it is worthwhile to step through the code and observe the comments. At this point, step over each instruction until you get to line 1098 in OLESRV1.CPP -- the call to OnFrameWindowActivate. Among other things, you will see the following tasks accomplished:

   - Get the document type used in SetActiveObject calls.

   - Get the in-place client-site interface.

   - See if the container wants to go in-place right now.

   - Get the parent window to create the COleIPFrameWnd.

   - Create the inplace frame window.

   - Send an activate notification to the container.

   - Get the frame and doc window interfaces as well as other information.

- Set up the shared menu.

- Allow the server to install frame controls in the container.

- Update the zoom factor information before creating control bars.

- Resize the window to match the object.

- Set the active object.

- Add the frame and the document-level frame controls.

- Show any hidden modeless dialogs.

- Attempt toolbar negotiation.

- Install the menu and a hook that forwards messages from the menu to the inplace frame window.

- Make sure the object is scrolled into view.

- Show the inplace frame window and set the focus.

As you can see, ActivateInPlace does a lot of work. It is also very RPC-intensive.

8. Step into the first line of OnFrameWindowActivate. Observe that this function sends the final notifications via the container to activate the server.

9. Press the F5 key or choose the GO button to finish executing OnFrameWindowActivate. Under normal circumstances, Scribble would come up already activated in place within Contain. However, in this case, you have already executed past the code that set the focus to the activated server. Rremember the call to pFrameWnd->Set Focus on line 1094 in the ActivateInPlace function in OLESRV1.CPP? By continuing to step through the code you have reset the focus back to the debugger. Therefore, you must now switch tasks manually to Contain.

10. Switch tasks to Contain. You will see Scribble's menu and toolbar within Contain, and you will be able to draw in the embedded item's rectangle.

Finishing the Debug Session

When you finish with Scribble, close the document window. Contain's menu and toolbar reappear, and the Scribble debugging session ends within the second instance of the debugger.

Although Scribble has terminated, the second instance of the debugger is still running. To avoid complications, terminate the second instance of the debugger. You need to do this because whenever you call into a server to embed a new item or activate an existing one, the debugger will start another instance to debug this server, even if it is the same server that

is already attached to another item in the document. Therefore, it is possible (but not desirable) to have multiple instances of the debugger debugging multiple instances of the same server all connected to the same container.

Once the second instance of the debugger has been terminated, it is safe to embed another Scrib Document object in Contain. It is not necessary to terminate the container before doing so.

Trying New Things in Future Debug Sessions

If you have more time in a subsequent debugging session, try stepping into (rather than over) some of the function calls. The flow of control goes back and forth between the container and the server many times, and the debugger will track this flow accurately, bringing you into container code, then server code, and so forth. Functions of interest include:

- CanInPlaceActivate
- OnInPlaceActivate
- GetWindow
- OnUIActivate
- GetWindowContext
- SetActiveObject
- SetMenu
- ShowObject

The server maintains several pointers (interfaces) into the container's code through which it calls the container's member functions. These pointers include m_lpClientSite, lpInPlaceSite, and pFrameWnd.

Additional reference words: kbinf 2.00
KBCategory: kbtool kbole kbinterop
KBSubcategory: WBDebug VCx86

Using Single/Multiple Instances of an OLE Object in MFC

PSS ID Number: Q141154

The information in this article applies to:

- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft Visual C++ for Windows, versions 1.0, 1.5, 1.51, 1.52
  - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY

This article explains how class factories are registered in an OLE local server created using MFC. This topic is discussed with respect to whether a single instance of an application can service multiple clients or whether a separate instance will be launched for each request made by a client.

The class factory specifically attached to the document template behaves differently based on whether it's a default SDI (single document interface) or an MDI (multiple document interface) application. Other generic class factories behave according to how they are registered with OLE.

MORE INFORMATION

MFC provides an interface implementation for the class factory (IClassFactory) in a class called COleObjectFactory, which serves as a generic type of factory. In addition, MFC also provides a specific class factory called COleTemplateServer, derived from COleObjectFactory, to create documents using the framework's document/view architecture.

In the case of a local server, the class factories have to be registered with OLE so that other applications can connect to them. In MFC, all the class factories of a server are typically registered in CWinApp::InitInstance by calling the static function COleObjectFactory::RegisterAll(). This function calls COleObjectFactory::Register() for each of the class factories, which in turn calls CoRegisterClassObject in the case of a local server. In the case of an inproc server, there is no need to register the class factories with OLE because COM loads the DLL (that is, the server) and invokes a well-known entry-point, called DllGetClassObject, to retrieve an IClassFactory interface.

When a client requests that a server create an instance of an object, the decision to use a running instance of the server, if one is available, or to launch another instance of the server is made by OLE depending on how the class factory for the object is registered. The determining factor is the fourth parameter to CoRegisterClassObject, which specifies how many objects can be created using this class factory. If REGCLS_SINGLEUSE is specified, OLE will launch another instance of the application each time a client requests an instance of an object class. If REGCLS_MULTIPLEUSE is specified, one instance of the application can service any number of

objects.

MDI applications use the same main frame window (that is, the same instance of the application) and create a new MDI child window for each request made by a client to create an object instance. When a client calls CoCreateInstance to create a document window object exposed by COleTemplateServer, a new MDI child window is created using the same MDI frame window. MFC implements this concept by passing in FALSE as the m_bMultiInstance parameter to the ConnectTemplate function of COleTemplateServer. This registers the class factory as single instance by passing REGCLS_MULTIPLEUSE to CoRegisterClassObject.

SDI applications, on the other hand, can manage only one document window at a time. Hence, SDI applications are by default single use only; that is, a separate instance (or a document window) is created for each call to CoCreateInstance made by a client. This is implemented in an MFC application by passing in TRUE as the m_bMultiInstance parameter to COleTemplateServer::ConnectTemplate, which registers the class factory as multiple instance by passing REGCLS_SINGLEUSE to CoRegisterClassObject.

Generic class factories that are used to create any C++ object are registered as single- or multiple-use based on how the constructor for COleObjectFactory is invoked. The constructor takes a bMultiInstance parameter that specifies whether the class factory is registered as REGCLS_SINGLEUSE (bMultiInstance == TRUE) or REGCLS_MULTIPLEUSE (bMultiInstance == FALSE).

When ClassWizard is used to add OLE-creatable classes to an MFC application, it adds the DECLARE_OLECREATE and IMPLEMENT_OLECREATE macros. DECLARE_OLECREATE declares a static object of type COleObjectFactory, and IMPLEMENT_OLECREATE creates this object by invoking the constructor of COleObjectFactory with the bMultiInstance parameter set to FALSE. In other words, ClassWizard registers the generic class factory as REGCLS_MULTIPLEUSE. Hence by default, a single instance of an OLE-creatable class generated by ClassWizard can serve multiple requests to create an instance of this object class.

This behavior can be altered by replacing IMPLEMENT_OLECREATE with a custom macro, say MY_IMPLEMENT_OLECREATE, which invokes the COleObjectFactory constructor with the bMultiInstance parameter set to TRUE, which will then register a REGCLS_SINGLEUSE class factory. For example:

```
#define MY_IMPLEMENT_OLECREATE(class_name, external_name, l, w1, w2,
        b1, b2, b3, b4, b5, b6, b7, b8) \
        COleObjectFactory class_name::factory(class_name::guid, \
        RUNTIME_CLASS(class_name), TRUE, _T(external_name)); \
        const GUID CDECL class_name::guid = \
        { l, w1, w2, { b1, b2, b3, b4, b5, b6, b7, b8 } }; \

MY_IMPLEMENT_OLECREATE(CMyClass, "Test", 0xc9c99ae0, 0xad41, 0x101b,
                       0x83, 0xea, 0x0, 0x0, 0x83, 0x78, 0xac, 0x8b)
```

The CLSID used in this macro call should be replaced with the actual CLSID assigned for this particular object, which can be found in the original IMPLEMENT_OLECREATE macro.

REFERENCES
==========

Inside OLE 2.0, Second Edition.

Visual C++ Books Online and the MFC source code for the functions mentioned in the article.

Additional reference words: kbinf 1.00 1.50 2.00 2.10 2.51 2.52 3.00 3.10 3.20 4.00
KBCategory: kbole
KBSubcategory: MfcOLE

Variant Handling in an MFC Server w/ VB 4.0 as Its Client

PSS ID Number: Q142223

The information in this article applies to:

- Standard and Professional Editions of Microsoft Visual Basic programming system for Windows, version 4.0

- The Microsoft Foundation Classes (MFC) included with: Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0

SUMMARY

It is possible for Visual Basic 4.0 to pass parameters to automation methods either by value (ByVal) or by reference (ByRef). For a method that expects a non-VARIANT parameter passed by value, MFC's implementation of IDispatch will coerce a parameter passed by reference to a value. However, this is not the case with an automation method expecting a VARIANT parameter. The MFC framework cannot coerce a VARIANT parameter because it does not know what type is expected by the method. This can cause problems in a method that expects a VARIANT parameter to be passed by value. If the parameter is passed by reference and the VARIANT is referenced directly, the value obtained from the variant will be incorrect. Hence, an automation method that expects a VARIANT to be passed by value should check whether it was passed a reference and coerce the VARIANT if necessary. This article illustrates how you could implement this.

MORE INFORMATION

Consider the following method named Add exposed by a MFC automation server with a ProgID TestVar.Document:

```
LPDISPATCH Add(const VARIANT FAR& varTest);
```

Given the following code in Visual Basic, it may pass the parameter either by reference or by value:

```
Dim doc As Object
Set doc = CreateObject("TestVar.Document")

Dim docDispatch As Object
Dim varParam As Variant
varParam = 2
```

Visual Basic will pass the parameter by reference in these cases:

```
Set docDispatch = doc.Add(varParam)
doc.Add varParam
```

Visual Basic will pass the parameter by value in these cases:

```
   Set docDispatch = doc.Add((varParam))
   doc.Add (varParam)
   Set docDispatch = doc.Add(2)
   doc.Add 2
```

From these examples, you may notice that Visual Basic will pass all variables by reference unless the () operator is used to indicate that the variable should be passed by value. Moreover, Visual Basic will pass all constants by value.

Because of the possibility of the parameter being passed by reference, it is necessary to coerce the parameter to a value before using it in the automation server. Making a copy of the VARIANT parameter using the VariantCopyInd function will perform the necessary indirection, if the source VARIANT parameter passed is specified to be VT_BYREF. The following code will properly handle a VARIANT passed either by reference or by value.

```
LPDISPATCH CTestVarDoc::Add(const VARIANT FAR& varTest)
{
    HRESULT hr;
    VARIANT var;
    VariantInit(&var);

    hr = VariantCopyInd(&var, (LPVARIANT)&varTest);

    if (FAILED(hr))
       return NULL;

    // Now use var instead of varTest
    ...
}
```

Additional reference words: kbinf 4.00 3.00 3.10 3.20 4.0 2.0 2.1 2.2
KBCategory: kbprg kbole kbcode
KBSubcategory: MfcOLE

VB Automation of Visual C++ Server Using OBJ1.OBJ2.prop Syntax

PSS ID Number: Q137343

The information in this article applies to:

- The Microsoft Foundation Classes included with:
    - Microsoft Visual C++ for Windows, versions 1.5, 1.51, 1.52
    - Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2, 4.0, 4.1

SUMMARY
=======

It is often necessary to access properties within nested objects exposed by OLE Automation. The Visual Basic syntax is:

```
OBJ1.OBJ2.Property
```

This article demonstrates the creation of a simple OLE Automation server with Visual C++, accessible from Visual Basic, that exposes such an object.

MORE INFORMATION
================

The following steps demonstrate how to create a Visual C++ OLE Automation server that allows Visual Basic to use the OBJ1.OBJ2.Property syntax.

Steps to Create the Visual C++ OLE Automation Server
---

1. Create an AppWizard-generated project called AutoServ with OLE Automation enabled.

2. Once the project has been generated, start ClassWizard.

3. Click the OLE Automation tab.

4. Click the Add Class button, enter the following values, and then click Create Class:

   ```
   Class Name:    nested
   Class Type:    CCmdTarget
   Check:         OLE Automation
   ```

5. Click the ClassWizard's OLE Automation tab. Set the Class Name to nested, click the Add Property button, enter the following values, and then click OK:

   ```
   External Name: Value
   Type:          long
   ```

6. Change the Class Name in ClassWizard to CAutoServDoc. If you are

prompted to save changes, click Yes. Then click the Add Method button, enter the following values, and click OK:

External Name: Nested
Return Type: LPDISPATCH

7. Click OK to accept the additions created by ClassWizard.

8. Open the project's AutoServDoc.h file, and add the following line to the beginning of the file:

```
#include "nested.h"
```

9. To the same file, add a public member variable m_nested of type nested. A pointer mechanism could have been used to maintain the nested class; however, for this example, the chosen method will automatically create and destroy the nested object within the Documents constructor and destructor respectively.

10. Open the project's nested.h file, and modify the class so that the constructor and destructor are public methods.

11. Open the projects AutoServDoc.cpp file, and modify the Nested Method as follows:

```
LPDISPATCH CAutoServDoc::Nested()
{
   //TODO: Add your dispatch handler code here
   return m_nested.GetIDispatch(TRUE);
}
```

12. Build the project, and then run AutoServ.exe to register the server.

Steps to Test the Server in Visual Basic

1. Start a new project in Visual Basic that will be used to test the OLE Automation server.

2. On the View menu, click Code.

3. In the Object combo box, select (general).

4. Enter the following code:

```
Dim Server As object
```

5. Select Form in the Object combo box. Modify the Sub procedure as follows:

```
Sub Form_Load ()
   Set Server = CreateObject("autoserv.document")
   Server.Nested.Value = 10
   x = Server.Nested.Value
End Sub
```

6. Add x and Server.Nested.Value to the Visual Basic watch window, so that you can observe the changes while stepping into the Visual Basic program.

Additional reference words: 1.50 1.51 1.52 2.00 2.10 2.20 2.50 2.51 2.52 3.00 3.10 3.20 4.00 4.10 vc vb client controller
KBCategory: kbole kbinterop kbprg kbcode
KBSubcategory: MfcOLE

Win32s OLE 16/32 Interoperability

PSS ID Number: Q123422

The information in this article applies to:

- Microsoft Win32s version 1.20

The OLE support provided in Win32s version 1.2 provides full 16/32 interoperabililty for local servers (EXE servers). Therefore, you can embed a 32-bit object implemented by a local server in a 16-bit container and vice-versa.

There is no built-in support in Win32s for 16/32 interoperability for in-process servers (DLL servers). However, you can use Universal Thunks to load a 16-bit DLL in the context of a 32-bit process. This allows you to embed a 16-bit object implemented by a DLL server in a 32-bit container. However, it is quite complicated to write this code because:

- Any OLE interface has a hidden "this" pointer which you must handle in your thunking code.
- OLE uses callbacks. If your 32-bit container calls IDataObject:DAdvise, then your 16-bit server may call back into the 32-bit side with the Advise interface. Your thunking code will have to handle this type of conversation.

NOTE: Embedding a 32-bit object implemented by an in-process server in a 16-bit container is supported under Windows NT 3.5. However, this functionality may not work correctly for your in-process server because IDispatch and any custom interfaces do not work.

Additional reference words: 1.20
KBCategory: kbole
KBSubcategory: W32s

16-Bit App WNetGetCaps Call Return Value on Win32

PSS ID Number: Q120359

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.5, 3.51, and 4.0

16-bit Windows-based applications often call WNetGetCaps() to determine the capabilities of the installed network. When a Windows-based application running on Windows NT calls WNetGetCaps(), the return value is 0x8004, which corresponds to WNNC_NET_Multinet | WNNC_SUBNET_WinWorkgroups.

However, the Windows NT Windows on Windows (WOW) layer and Windows 95 do not support the Windows for Workgroups Multinet (MNet) APIs, so a call to one of these APIs returns a failed Dynalink error.

The return value of WNetGetCaps() may not seem technically correct for Windows for Workgroups. It was designed to be compatible with all existing 16-bit Windows-based applications.

If you need to determine whether a 16-bit Windows-based application is running on Windows NT or MS-DOS/Windows version 3.1, use GetWinFlags(). GetWinFlags() returns a WF_WINNT flag if the application is running under WOW on Windows NT.

GetWinFlags() is an existing function that was modified in WOW to add the following flag:

```
   #define WF_WINNT         0x4000
```

Additional reference words: 3.50 4.00
KBCategory: kbprg
KBSubcategory: SubSys

32-Bit Scroll Ranges

PSS ID Number: Q104311

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

You can use 32-bit scroll ranges by calling GetScrollPos(); however, you cannot get 32-bit positions for notifications sent while thumb tracking, that is, via the SB_THUMBPOSITION message. This is because thumb position information is not queryable via an application programming interface (API). You only can obtain the 32-bit scroll information only before or after the scroll has taken place.

The scroll bar APIs allow setting a scroll range up to 0x7FFFFFFF via SetScrollRange(), and setting a scroll position within that range using SetScrollPos(). If the WM_HSCROLL or WM_VSCROLL message is processed, the information returned for scroll bar position, nPos, is only a 16-bit value. To obtain the 32-bit information, the GetScrollPos() API must be used.

Additional reference words: 3.10 3.50 scrollbar
KBCategory: kbui
KBSubcategory: UsrCtl

Accessing Parent Window's Menu from Child Window w/ focus

PSS ID Number: Q92527

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In an MDI-like application, the user must be allowed to pull down menus in the parent window by using menu mnemonics even though the child window or one of its children may have the focus. This can be done by creating child windows without a system menu or by processing the WM_MENUCHAR and WM_SYSCOMMAND/SC_KEYMENU messages to programatically pull down the parent's menu.

MORE INFORMATION

If a child window with a system menu has the focus and the user attempts to access the parent's menu with the keyboard using the menu mnemonic (ALT+mnemonic character), Windows will beep and the parent's menu will not be pulled down. This problem occurs because the parent window does not have the focus and because the window with the focus does not have a menu corresponding to the mnemonic. (Child windows cannot have menus other than the system menu.)

If the child window with the focus does not have a system menu, Windows assumes that the menu mnemonic is for the nearest ancestor with a system menu and passes the message to that parent. Consequently, it is possible to use menu mnemonics to pull down a parent's menu if the descendant windows do not have system menus.

If the child window with the focus has a system menu, Windows will beep if a menu mnemonic corresponding to a parent menu is typed. This can be prevented and the parent menu can be dropped down using the following code in the window procedure of the child window:

```
 case WM_MENUCHAR:
    PostMessage(hwndWindowWithMenu, WM_SYSCOMMAND, SC_KEYMENU, wParam);
    return(MAKELRESULT(0, 1));
```

WM_MENUCHAR is sent to the child window when the user presses a key sequence that does not match any of the predefined mnemonics in the current menu. wParam contains the mnemonic character. The child window posts a WM_SYSCOMMAND/SC_KEYMENU message to the parent whose menu is to be dropped down, with lParam set to the character that corresponds to the menu

mnemonic.

The above code can also be used if the child window with the focus does not have a system menu but an intermediate child window with a system menu exists between the child with the focus and the ancestor whose menu is to be dropped. In this case, the code would be placed in the intermediate window's window procedure.

Additional reference words: 3.10 3.00 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMen

Accessing the Application Desktop from a Service

PSS ID Number: Q115825

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51

SUMMARY

=======

Under Windows NT, version 3.1, if you want a service to have access to the application desktop, you must run the service in the LocalSystem account. A service process running in the LocalSystem account (or a process started from such a service) can display message boxes, windows, and dialog boxes. Processes that are running in the LocalSystem account are not terminated by the system during logoff. A number of changes were made to Windows NT, version 3.5, that affect the way Windows NT interacts with these services. In addition, Windows NT 3.51 has a richer set of desktop APIs.

NOTE: Running interactive services under the system account is a VERY dangerous practice. This is especially true of the command processor and batch files. A user who wants to control the system can just hit CTRL+C to get an interactive system command prompt.

MORE INFORMATION

================

The following are new features of Windows NT, version 3.5, that affect services:

- The account of the logged in user is the only account granted access to the application desktop. The LocalSystem no longer has access. Therefore, it is possible to get access to the desktop by impersonating the user before making any USER or GDI calls.

- Console and GUI applications started from a service process during a particular logon session are run on an invisible window station and desktop that are unique to that session. The window station and desktop are created automatically when the first application in the session starts; they are destroyed when the last application exits. There is no way to make these invisible desktops visible.

- If you want a service in the localsystem account to interact with the logged-on user, specify the SERVICE_INTERACTIVE_PROCESS flag in the call to CreateService(). For example:

```
schService = CreateService(
   schSCManager,
   serviceName,
   serviceName,
   SERVICE_ALL_ACCESS,
```

```
        SERVICE_INTERACTIVE_PROCESS | SERVICE_WIN32_OWN_PROCESS,
        SERVICE_DEMAND_START,
        SERVICE_ERROR_NORMAL,
        lpszBinaryPathName,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL );
```

If you specify an account other than localsystem when using SERVICE_INTERACTIVE_PROCESS, you will get error INVALID_PARAMETER (87).

- If you use CreateProcess() to launch your process and you want your service to log onto the users desktop, assign the lpdesktop parameter of the STARTUPINFO struct with "WinSta0\\Default".

- Services that simply need a visible user notification can do this by calling MessageBox() with the MB_SERVICE_NOTIFICATION flag. Using the MB_DEFAULT_DESKTOP_ONLY flag works as well, but only if the user's desktop is active. If the workstation is locked or a screen saver is running, the call will fail.

  NOTE: If you are writing code for an application that can be run as either a service or an executable, you can't use MB_SERVICE_NOTIFICATION as well as a non-NULL hwndOwner.

- Any output done to a window is not displayed or made available to the application in any way. Attempts to read bits from the display results in a failure.

- GUI services do not receive WM_QUERYENDSESSION/WM_ENDSESSION messages at logoff and shutdown; instead, they receive CTRL_LOGOFF_EVENT and CTRL_SHUTDOWN_EVENT events. These services are not terminated by the system at logoff.

Additional reference words: 3.50
KBCategory: kbprg
KBSubcategory: BseService

Accessing the Event Logs

PSS ID Number: Q108230

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

Event logs are used to store significant events, such as warnings, errors, or information. There are five operations that can be performed on event logs through the event logging application programming interface (API): backup, clear, query, read, and write.

The default event logs are the Application event log, the Security event log, and the System event log. Access to these event logs is determined by which account the application is running under.

MORE INFORMATION

================

The following table shows which accounts are granted access to which logs and what type of access is granted under Windows NT 3.1:

Log	Account	Access Granted
Application	LocalSys	read write clear
	Admins	read write clear
	ServerOp	read write clear
	World	read write
Security	LocalSys	read write clear
	Admins	read clear
System	LocalSys	read write clear
	Admins	read clear
	ServerOp	read clear
	World	read

Table 1 - access granted in Windows NT 3.1

The Local System account (LocalSys) is a special account that may be used by Windows NT services. The Administrator account (Admins) consists of the administrators for the system. The Server Operator account (ServerOp) consists of the administrators of the domain server. The World account includes all users on all systems.

Changes made were for Windows NT 3.5:

Log	Account	Access Granted

Application	LocalSys	read write	clear
	Admins	read write	clear
	ServerOp	read write	clear
	World	read write	
Security	LocalSys	read write	clear
	Admins	read	clear
	World	read	clear *
System	LocalSys	read write	clear
	Admins	read write	clear **
	ServerOp	read	clear
	World	read	

Table 2 - access granted under Windows NT 3.5

 * Users that have been granted manage auditing and security log rights
   can read and clear the Security log.

** Admins can write to the System log.

The following table shows which types of access are required for the
corresponding event logging API:

Event Logging API	Access Required
OpenEventLog()	read
OpenBackupEventLog()	read
RegisterEventSource()	write
ClearEventLog()	clear

Table 3 - access required for event logging APIs

As an example, OpenEventLog() requires read access (see Table 2). A member
of the ServerOp account can call OpenEventLog() for the Application event
log and the System event log, because ServerOp has read access for both of
these logs (see Table 1). However, a member of the ServerOp account cannot
call OpenEventLog() for the Security log, because it does not have read
access for this log (see Table 1).

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Accessing the Macintosh Resource Fork

PSS ID Number: Q106663

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

Resource forks are implemented as NTFS streams named AFP_Resource. There is no structure to the fork; it is exactly whatever the Macintosh writes to it. The resource fork can be written to using standard Win32 application programming interfaces (APIs). Refer to the forks as <FileName>:AFP_Resource.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Accurately Showing on the Screen What Will Print

PSS ID Number: Q75469

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

Many applications have an option where the screen display is set to closely correspond to the printed output. This article discusses some of the issues involved in implementing this feature.

MORE INFORMATION
================

If a screen font is available that exactly matches (or at least very closely corresponds to) the chosen printer font, then the process is very straightforward and consists of seven steps:

1. Retrieve a device context (DC) or an information context (IC) for the printer.

2. Call EnumFontFamilies() to obtain a LOGFONT structure for the chosen printer font. The nFontType parameter to the EnumFontFamilies() callback function specifies if a given font is a device font.

3. Get a DC for the screen.

4. Convert the lfHeight and lfWidth members of the LOGFONT structure from printer resolution units to screen resolution units. If a mapping mode other than MM_TEXT is used, round-off error may occur.

5. Call CreateFontIndirect() with the LOGFONT structure.

6. Call SelectObject(). GDI will select the appropriate screen font to match the printer font.

7. Release the printer device context or information context and the screen device context.

If a screen font that corresponds to the selected printer font is not available, the process is more difficult. It is possible to modify the character placement on the screen to match the printer font to show justification, line breaks, and page layout. However, visual similarity between the printer fonts and screen fonts depends on a

number of factors, including the number and variety of screen fonts available, the selected printer font, and how the printer driver describes the font. For example, if the printer has a serifed Roman-style font, one of the GDI serifed Roman-style fonts will appear to be very similar to the printer font. However, if the printer has a decorative Old English-style font, no corresponding screen font will typically be available. The closest available match would not be very similar.

To have a screen font that matches the character placement of a printer font, do the following:

1. Perform the seven steps above to retrieve an appropriate screen font.

2. Get the character width from the TEXTMETRIC structure returned by the EnumFonts function in step 2 above. Use this information to calculate the page position of each character to be printed in the printer font.

3. Allocate a block of memory and specify the spacing between characters. Make sure that this information is in screen resolution units.

4. Specify the address of the memory block as the lpDx parameter to ExtTextOut(). GDI will space the characters as listed in the array.

Additional reference words: 3.00 3.10 3.50 4.00 95 WYSIWYG
KBCategory: kbprint
KBSubcategory: GdiPrn

Action of Static Text Controls with Mnemonics

PSS ID Number: Q65883

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The text of a static control may contain a mnemonic, which is a character with which the user can access the control. A mnemonic is indicated to the user by underlining the character in the text of the control, and is created by preceding the desired character with an ampersand (&).

Mnemonic characters are used in conjunction with the ALT key to allow quick access to a control with the keyboard. When the user enters the key combination of the ALT key and the mnemonic character, Windows sets the input focus to the corresponding control and performs the same action as when the mouse is clicked on that control. Push buttons, option buttons, and check boxes all behave in this manner.

Because static text controls do not accept the focus, the behavior of a mnemonic in a static text control is different. When the user enters the mnemonic of a static text control, the focus is set to the next enabled nonstatic control. A static text control with a mnemonic is primarily used to label an edit control or list box. When the user enters the mnemonic, the corresponding control gains the focus.

In this context, the order in which windows are created is important. In a dialog box template, the control defined on the line following the static text control is considered to be "next."

When the user enters the mnemonic of a static text control and the next control is either another static text control or a disabled control, Windows searches for a control that is nonstatic and enabled. In some cases, it may be preferable to disable the mnemonic of a static text control when the control it labels is also disabled. For more information, please query in the Microsoft Knowledge Base on the following word:

   mnemonic

MORE INFORMATION
================

The dialog box described by the following dialog box template might be

displayed by an application when the user chooses Open from the File menu:

```
IDD_FILEOPEN DIALOG LOADONCALL MOVEABLE DISCARDABLE 9, 22, 178, 112
CAPTION "File Open..."
STYLE WS_CAPTION | DS_MODALFRAME | WS_SYSMENU | WS_VISIBLE | WS_POPUP
BEGIN
   CONTROL "File&name:", ID_NULL, "static",
           SS_LEFT | WS_GROUP | WS_CHILD, 5, 5, 33, 8
   CONTROL "", ID_NAMEEDIT, "edit",
           ES_LEFT | ES_AUTOHSCROLL | WS_BORDER | WS_TABSTOP
           | WS_CHILD | ES_OEMCONVERT, 40, 4, 90, 12
   CONTROL "Directory:", ID_NULL, "static", SS_LEFT | WS_CHILD,
           5, 20, 35, 8
   CONTROL "", ID_PATH, "static", SS_LEFT | WS_CHILD, 40, 20, 91, 8
   CONTROL "&Files:", ID_NULL, "static", SS_LEFT | WS_GROUP
           | WS_CHILD, 5, 33, 21, 8
   CONTROL "", ID_FILELIST, "listbox", LBS_NOTIFY | LBS_SORT
           | LBS_STANDARD | LBS_HASSTRINGS | WS_BORDER | WS_VSCROLL
           | WS_TABSTOP | WS_CHILD, 5, 43, 66, 65
   CONTROL "&Directories:", ID_NULL, "static", SS_LEFT | WS_GROUP
           | WS_CHILD, 75, 33, 49, 8
   CONTROL "", ID_DIRLIST, "listbox", LBS_NOTIFY | LBS_SORT
           | LBS_STANDARD | LBS_HASSTRINGS | WS_BORDER | WS_VSCROLL
           | WS_TABSTOP | WS_CHILD, 75, 43, 65, 65
   CONTROL "OK", IDOK, "button", BS_DEFPUSHBUTTON | WS_TABSTOP
           | WS_CHILD, 139, 4, 35, 14
   CONTROL "Cancel", IDCANCEL, "button", BS_PUSHBUTTON | WS_TABSTOP
           | WS_CHILD, 139, 23, 35, 14
END
```

In this dialog box, one static text control, with identifier ID_PATH, is used to display the current path. The other four static text controls label other controls, as follows:

```
   "File&name"         labels the ID_NAMEEDIT edit control
   "Directory"         labels the ID_PATH static control display
   "&Files"            labels the ID_FILELIST list box
   "&Directories"      labels the ID_DIRLIST list box
```

When the user enters the key combination ALT+N, Windows sets the focus to the edit control identified in the dialog template as ID_NAMEEDIT, because it is the next enabled nonstatic control. If that edit control was disabled by the EnableWindow function, pressing ALT+N would move the focus to the next enabled nonstatic control. This control would be the list box identified as ID_FILELIST.

Note that the static control "Directory" has no mnemonic; therefore, keyboard input does not affect it.

When the user enters ALT+F, the focus moves to the ID_FILELIST list box, if it is enabled. In the same manner, ALT+D moves the focus to the ID_DIRLIST list box.

If ID_DIRBOX is disabled, ALT+D moves the focus to the OK button, the

next enabled nonstatic control. Windows treats this as if the user pressed and released the mouse button over the OK button. For more information on how to prevent this behavior, query the Microsoft Knowledge Base on the following word:

   mnemonic

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 radio shortcut
KBCategory: kbui
KBSubcategory: UsrCtl

Adding a Custom Template to a Common Dialog Box

PSS ID Number: Q86720

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

SUMMARY

=======

Many applications developed for the Microsoft Windows environment using dialog boxes from the common dialogs library (COMMDLG.DLL) require custom dialog templates. An application generally uses a custom dialog box template to add controls to a standard common dialog box. The text below discusses the steps required to implement a custom dialog box template with a common dialog box.

A custom dialog box template is most often used in conjunction with a hook function. For details on using a hook function with one of the common dialog boxes, query on the following words in the Microsoft Knowledge Base:

   steps adding hook function

MORE INFORMATION

================

CDDEMO, one of the advanced sample applications provided with version 3.1 of the Microsoft Windows Software Development Kit (SDK), demonstrates adding a hook function to the File Open dialog box. The five steps required to modify the CDDEMO application to use a custom dialog box template in its File Open dialog box are as follows:

1. Edit the FILEOPEN.DLG template in the Windows SDK advanced sample applications directory (by default, C:\WINDEV\SAMPLES\COMMDLG). All existing controls must remain in the dialog template; add additional controls, if desired. To demonstrate the process, make a copy of the FILEOPEN.DLG template and include it in the CDDEMO.RC file. Modify the title of the "Cancel" button to "CANCEL." Renaming the button minimizes the potential for error while demonstrating that the application loaded the custom dialog box template.

2. In the application, modify the Flags member of the OPENFILENAME data structure to include the OFN_ENABLETEMPLATE initialization flag.

3. Specify MAKEINTRESOURCE(FILEOPENORD) as the value of the lpTemplateName member of the OPENFILENAME data structure.

4. Specify ghInst as the value of the hInstance member of the

OPENFILENAME data structure.

5. Use the #include directive to include DLGS.H in the CDDEMO.RC file.

If an application adds a hook function to a common dialog box, the hook receives all messages addressed to the dialog box. With the exception of the WM_INITDIALOG message, the hook function receives messages before its associated common dialog box does. If the hook function processes a message completely, it returns TRUE. If the common dialog box must provide default processing for a message, the hook function returns FALSE.

In the hook function, the application should process messages for any new controls added through the custom dialog box template. If the standard common dialog box template contains a control that is unnecessary in a particular application, hide the control when the hook function processes the WM_INITDIALOG message. Use the ShowWindow() API to hide a control; do not delete any controls from the common dialog box template. To indicate that the common dialog boxes DLL does not function properly if any controls are missing, the debug version of Windows displays FatalExit 0x0007.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Adding a Hook Function to a Common Dialog Box

PSS ID Number: Q86721

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Many applications developed for the Microsoft Windows environment using dialog boxes from the common dialogs library (COMMDLG.DLL) require hook functions. A hook function for one of the common dialog boxes is similar to a subclass procedure for a standard Window control, such as an edit control. Through a hook function, an application can process all messages addressed to the dialog box. The text below discusses the steps required to implement a hook function with a common dialog box.

A hook function is most often used in conjunction with a custom dialog template. For details using a custom dialog template with one of the common dialog boxes, query on the following words in the Microsoft Knowledge Base:

   steps add custom template

MORE INFORMATION

The hook function receives all messages addressed to a common dialog box. With the exception of the WM_INITDIALOG message, the hook function receives messages before its associated common dialog box does. If the hook function processes a message completely, it returns TRUE. If the common dialog box must provide default processing for a message, the hook function returns FALSE.

CDDEMO, one of the advanced sample applications provided with version 3.1 of the Microsoft Windows Software Development Kit (SDK), demonstrates adding a hook function to the File Open dialog box. The eight steps involved in this process are as follows:

1. Add the standard common dialog box to the application without the hook function.

2. In the application's module definition (DEF) file, list the hook procedure name (for example, MyHookProc) in the EXPORTS section.

3. Define a FARPROC variable (for example, lpfnHookProc)

4. In the application, before completing the OPENFILENAME data structure, call the MakeProcInstance function to create a procedure instance address for the hook procedure.

5. Set the lpfnHook member of the OPENFILENAME data structure to the procedure address of the hook function.

6. Specify OFN_ENABLEHOOK as one of the initialization flags in the Flags member of the OPENFILENAME structure.

7. Code the hook function to process messages as required. A sample hook function follows below.

8. After the user dismisses the common dialog box, call the FreeProcInstance function to free the procedure instance address.

The following code is a sample hook function:

```
BOOL FAR PASCAL MyHookProc(HWND hDlg, unsigned message,
                           WORD wParam, LONG lParam)
{
   switch (message)
   {
      case WM_INITDIALOG:
         OutputDebugString("Hello hook function!");
         return TRUE;

      case WM_COMMAND:
         switch(wParam)
         {
            case IDD_MYNEWCONTROL:
            // Perform appropriate processing here...
               return TRUE;

            default:
               break;
         }
         break;

      default:
         break;
   }
   return FALSE;
}
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Adding Categories for Events

PSS ID Number: Q115947

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

=======

The ReportEvent() API accepts a category ID as one of the arguments. The "Microsoft Win32 Programmer's Reference" states that you can add your own categories for events. This article shows you how to add categories that will be recognized by the Event Viewer; however, the article assumes that you already know how to create a message file and add an event source to the registry. For information about basic event logging, please see the logging sample in the Q_A\SAMPLES\logging directory on the "Win32 SDK" CD.

NOTE: The logging example that comes with Visual C++ 2.0 does not bind the MESSAGES.RC to the MESSAGES.DLL unlike the logging example that accompanies the Win32 SDK or NSDN Level 2. Binding MESSAGES.RC to the MESSAGES.DLL can be accomplished by adding MESSAGES.RC to MESSAGES.MAK.

NOTE: If you notice that there are some entries that have a .DLL name and a driver name while you are attempting to read messages from the event log, this means that the event message source has more than one message file. This means you need to parse the string and load each message file.

MORE INFORMATION

================

Just like events, category IDs are simply IDs in message file resources. However, in order to use categories, the following two requirements must be met:

- The category IDs must be sequentially numbered, starting with a message ID of 1.
- The event source entry in the registry must specify the category message file and the number of categories in the message file.

The first requirement is simply a matter of setting the MessageID entries in the message file for the categories. If all of your categories are listed at the top of the message file, you can assign the ID of 1 to the first message. Each message after that automatically gets the next ID value unless you specify otherwise in the MessageID entry.

The category entries in the registry are made by adding values to your event key. Normally, your event log application key already contains EventMessageFile and TypesSupported entries. You should add the following two entries:

- CategoryMessageFile

- CategoryCount

The CategoryMessageFile entry is of type REG_EXPAND_SZ. It should be set to the full path to the message file that contains the categories.

The CategoryCount entry is a REG_DWORD type. You should set this entry to the number of categories in the message file specified in CategoryMessageFile.

REFERENCES
==========

"Microsoft Win32 Programmer's Reference," Microsoft Corporation.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Adding Custom Error Strings to an MCI Device Driver

PSS ID Number: Q76411

The information in this article applies to:

- Microsoft Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY
=======

In the Microsoft Windows graphical environment, an application can use mciGetErrorString() to obtain the string associated with an error code returned from a media control interface (MCI) device driver.

An MCI driver can use a STRINGTABLE resource to store its error strings. The identifier constant for each string should correspond to the error value returned by the DriverProc function. This value must be greater than or equal to the constant MCIERR_CUSTOM_DRIVER_BASE to avoid confusion with the predefined MCI error codes.

MORE INFORMATION
================

For example, the Pioneer LaserDisc device driver, MCIPIONR.DRV, included with the Microsoft Device Development Kit (DDK) for Windows 3.1, contains the following declarations in its header file, MCIPIONR.H:

```
#define MCIERR_PIONEER_ILLEGAL_FOR_CLV  (MCIERR_CUSTOM_DRIVER_BASE)
#define MCIERR_PIONEER_NOT_SPINNING     (MCIERR_CUSTOM_DRIVER_BASE+1)
#define MCIERR_PIONEER_NO_CHAPTERS      (MCIERR_CUSTOM_DRIVER_BASE+2)
#define MCIERR_PIONEER_NO_TIMERS        (MCIERR_CUSTOM_DRIVER_BASE+3)
```

Its resource file, MCIPIONR.RC, contains the following STRINGTABLE definition:

```
STRINGTABLE
BEGIN
  MCIERR_PIONEER_ILLEGAL_FOR_CLV, "Illegal operation for CLV type disc."
  MCIERR_PIONEER_NOT_SPINNING, "The disc must be spun up to perform this \
operation."
  MCIERR_PIONEER_NO_CHAPTERS, "Chapters are not supported for this disc."
  MCIERR_PIONEER_NO_TIMERS, "All timers are in use. Cannot enable \
notification."
END
```

When mciGetErrorString() receives a value greater than or equal to MCIERR_CUSTOM_DRIVER_BASE, it looks in the driver's resource table for a string with the corresponding identifier.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbmm kbprg

KBSubcategory: MMMisc

Adding Point Sizes to the ChooseFont() Common Dialog Box

PSS ID Number: Q99668

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

When a TrueType font (or any other scalable font) is selected in the ChooseFont() common dialog box, a list of reasonable point sizes is displayed for selection. In some cases it is necessary to change this list to allow fewer or more selections.

The initial list of point sizes is hard-coded into COMMDLG.DLL but can be changed programmatically using a common dialog box hook function.

MORE INFORMATION

Most scalable fonts can be created at nearly any point size. Some TrueType fonts can be sized from 4 points to 999 points. A complete list of available point sizes for a font of this type would contain about 1000 elements, which can be prohibitively long and time-consuming to construct.

The ChooseFont() common dialog box attempts to limit the selection by listing only a few of the available point sizes in the size selection combo box.

In certain cases, it may be desirable to offer more point-size selections in this dialog box. In this case, a common dialog box hook procedure can be used to insert point sizes when a specific font is selected.

The following steps describe a technique that will allow new point sizes to be added to the size selection combo box. Pay special attention to step number 4:

1. In your common dialog box hook procedure, look for WM_COMMAND messages with wParam equal to the font name combo box (which is "cmb1" for Windows version 3.1).

2. When you get this message(s), check for the font name you are looking for (for example, you can compare the current selection to "Courier New"). If it's not the font you want, return.

3. If this is the font you want, post yourself a user-defined message. In response to this message, add the new point size to the Point Size combo box (which is "cmb3" for Windows 3.1). It's a good idea to double-check here that the point size you are adding isn't already in the combo box (so you don't get duplicates).

4. Once you add the new point size, set the item data for the new item equal to the point size you are adding. For example, if you are adding the string "15" to the combo box, you need to set the item data of this new item to 15.

The following code fragment demonstrates the above steps:

```
// Common Dialog ChooseFont() hook procedure.

UINT CALLBACK __export FontHook(HWND hwnd, UINT wm, WPARAM wParam,
LPARAM lParam)
{
   char szBuf[150];
   DWORD dwIndex;

   switch(wm)
   {
      case WM_COMMAND:
      // See if the notification is for the "Font name" combo box.
         if (wParam == cmb1)
         {
            switch (HIWORD(lParam))
            {
               case CBN_SETFOCUS:
               case CBN_KILLFOCUS:
                  break;
               default:
               // Check to see if it's the font we're looking for.
                  dwIndex = SendDlgItemMessage(hwnd, cmb1, CB_GETCURSEL,
                                               0, 0);
                  if (dwIndex != CB_ERR)
                     SendDlgItemMessage(hwnd, cmb1, CB_GETLBTEXT, (WPARAM)
                                        dwIndex, (LPARAM) ((LPSTR) szBuf));

               // Compare list box contents to the font we are looking for.
               // In this case, it's "Courier New".
                  if (strcmp(szBuf, "Courier New") == 0)
                  // It's the font we want. Post ourselves a message.
                     PostMessage(hwnd, WM_ADDNEWPOINTSIZES, 0, 0L);
            }
         }
      break;

      case WM_ADDNEWPOINTSIZES:
      // First look to see if we've already added point sizes to this
      // combo box.
         if (SendDlgItemMessage(hwnd, cmb3, CB_FINDSTRING, -1,
                                (LPARAM)(LPCSTR)"6") == CB_ERR)
         {
```

```
         // Not found, add new point size.
            dwIndex = SendDlgItemMessage(hwnd, cmb3, CB_INSERTSTRING,
                                         0, (LPARAM)(LPSTR)"6");

         // Also set the item data equal to the point size.
            SendDlgItemMessage(hwnd, cmb3, CB_SETITEMDATA,
                               (WPARAM)dwIndex, 6);
         }
         return TRUE; // Don't pass this message on.
   }
   return FALSE;
}
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Adding to or Removing Windows from the Task List

PSS ID Number: Q99800

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

There are no functions included in the Windows Software Development Kit (SDK) to add or remove windows from the task list. All top-level windows (that is, windows without parents) that are visible will automatically appear in the task list. A window can be removed from the task list by making it hidden. Call ShowWindow() with the SW_HIDE parameter to hide the window. To make it visible again, call ShowWindow() with the appropriate parameter such as SW_SHOW or SW_SHOWNORMAL.

Additional reference words: 3.00 3.10 3.50 tasklist
KBCategory: kbui
KBSubcategory: UsrMisc

Additional Information About GetTypeByName

PSS ID Number: Q138038

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0

SUMMARY
=======

GetTypeByName takes a service name in string format and returns a global unique identifier (GUID) for it. GetTypeByName is documented in the Win32 SDK. This article is an addendum to the documentation.

MORE INFORMATION
================

The purpose of GetTypeByName is to retrieve a GUID of a service. It does this first by examining the registry, looking for a local service. If the key does not exist, the function conducts a search of well-known TCP and UDP service types. If the service GUID is still unknown, a final comparison is made against a hard-coded list. See Srvguid.h for a list of most of the known GUIDs.

All comparisons made by GetTypeByName are case-sensitive.

GetTypeByName is exported from Wsock32.dll. To use it for anything other than registry information retrieval, call WSAStartup first. If WSAStartup is not called prior to calling GetTypeByName, the function fails, and GetLastError returns ERROR_SERVICE_DOES_NOT_EXIST, not WSANOTINITIALIZED.

REFERENCES
==========

"Win32 Software Development Kit Programmer's Reference," version 3.51.

Additional reference words: 4.00 GetTypeByName Windows 95
KBCategory: kbnetwork kbprg
KBSubcategory:

Additional Information for WIN32_FIND_DATA

PSS ID Number: Q120697

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

This article contains additional information about the WIN32_FIND_DATA members.

The WIN32_FIND_DATA structure contains three members that store the creation, last access, and last write time of a file. The time format for these three members (ftCreationTime, ftLastAccessTime, and ftLastWriteTime) are expressed in the Universal Time Convention (UTC). These three data members can be converted from UTC time to local time by calling the FileTimeToLocalFileTime api.

The WIN32_FIND_DATA structure contains two members that store the file size: nFileSizeHigh and nFileSizeLow. They are described as being the high and low order words of the size, but they are actually DWORDs. Therefore, nFileSizeHigh will be zero unless the file size is greater than 0xffffffff (4.2 Gig).

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

Additional Remote Debugging Requirement

PSS ID Number: Q106066

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

The printed and online documentation for remote debugging with WinDbgRm fail to mention one requirement. The binaries must be in the same drive and directory on both the target machine and the development machine.

WinDbg also expects to find the source files in the same directory in which the the binary file was built, but will browse for the source if it is not found in this location. WinDbg will automatically locate the source if the files are specified to the compiler with fully qualified paths.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsWindbg

Administrator Access to Files

PSS ID Number: Q102099

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51

A user that is a member of the Administrator group is not automatically granted access to any file on the local machine. For an administrator to access a file, permission must be specifically granted (as for any user) in the file's discretionary access control list (DACL).

If an administrator wants to access a file that he or she is not granted access to, the administrator must first take ownership of that file. Once ownership is taken, the administrator will have full access to the file. It is important to note that administrator cannot give ownership back to the original owner. If this were so, the administrator could take ownership of a file, examine it, and then assign it back to the original owner without that owner's knowledge.

NOTE: Because administrators have backup privileges, an administrator could back up a file (or entire volume) and restore it onto another system. The administrator could then take ownership of a file on this new system, examine it, and then restore from backup with original permissions, without the owner's knowledge. Please keep this in mind when thinking about file security.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Advanced Graphics Settings Slider under Windows 95

PSS ID Number: Q127066

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

The following table shows exactly what the Advanced Graphics Settings Slider control does:

Slider Control	Hardware Cursor	Memory Mapped I/O	Accelerated Functions
None	No	No	Safe mode level 2
Basic	No	No	Safe mode level 1
Most	No	Yes	All
Full	Yes	Yes	All

NOTES:

- The hardware cursor adjusts the SWCursor switch in SYSTEM.INI [display] section. It applies only to S3 and WD drivers.

- The Memory Mapped I/O adjusts the MMIO switch in SYSTEM.INI [display] section. It applies only to the S3 driver.

- Accelerated Functions adjusts the SafeMode switch in WIN.INI [Windows] section. It applies to all DIB engine minidrivers. Level 1 allows basic, safe acceleration, such as srcCopy bitblt, patblt, and so on. Level 2 completely bypasses the driver for all operations (except cursor).

Additional reference words: 4.00 95 Video SYSTEM Applet
KBCategory: kbui
KBSubcategory: UsrCtl

Advantages of Device-Dependent Bitmaps

PSS ID Number: Q94918

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

A DDB (device-dependent bitmap) is much faster than a DIB (device independent-bitmap) to BitBlt(). For this reason, it is often a good strategy under Win32 (as well as under Windows 3.1) to create a DDB from a DIB when caching or calling *Blt() functions.

The slight drawback of memory overhead for the DDB is handled well by Win32. Under Windows 3.1, the DDB memory could be marked as discardable. Under Win32, the memory will be paged out if system resources become tight (at least until the next repaint); if the memory is marked as PAGE_READONLY, it can be efficiently reused, [see VirtualProtect() in the Win32 application programming interface (API) Help file].

However, saving the DDB to disk as a mechanism for transfer to other applications or for later display (another invocation) is not recommended. This is because DDBs are driver and driver version dependent. DDBs do not have header information, which is needed for proper translation if passed to another driver or, potentially, to a later version of the driver for the same card.

MORE INFORMATION
================

Windows 95 and Windows NT 3.5 and later support DIBSections. DIBSections are the fastest and easiest to manipulate, giving the speed of DDBs with direct access to the DIB bits. NOTE: Win32s does not support DIBSections.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbgraphic
KBSubcategory: GdiBmp

Allocating and Using Class and Window Extra Bytes

PSS ID Number: Q34611

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The WNDCLASS structure contains two fields, cbClsExtra and cbWndExtra, which can be used to specify a number of additional bytes of memory to be allocated to the class structure itself or to each window created using that class.

MORE INFORMATION

Every application that uses class extra bytes and window extra bytes must specify the appropriate number of bytes before the window class is registered. If no bytes are specified, an attempt to store information in extra bytes will cause the application to write into some random portion of Windows memory, causing data corruption.

Windows version 3.1 will FatalExit if extra bytes are used improperly.

If an application does not use class extra bytes or window extra bytes, it is important that the cbClsExtra and cbWndExtra fields be set to zero.

Class and window extra bytes are a scarce resource. If more than 4 extra bytes are required, use the GlobalAlloc fucntion to allocate a block of memory and store the handle in class or window extra bytes.

Class Extra Bytes

For example, setting the value of the cbClsExtra field to 4 will cause 4 extra bytes to be added to the end of the class structure when the class is registered. This memory is accessible by all windows of that class. The number of additional bytes allocated to a window's class can be retrieved through the following call to the GetClassWord function:

```
nClassExtraBytes = GetClassWord(hWnd, GCW_CBCLSEXTRA);
```

The additional memory can be accessed one word at a time by specifying an offset, in BYTES (starting at 0), as the nIndex parameter in calls

to the GetClassWord function. These values can be set using the SetClassWord function.

The GetClassLong and SetClassLong functions perform in a similar manner and get or set four bytes of memory respectively:

```
   nClassExtraBytes = GetClassLong(hWnd, GCL_CBCLSEXTRA);
```

NOTE: A Win32-based application should use GetClassLong and SetClassLong, because the GCW_ indices are obsolete under Win32.

Window Extra Bytes

Assigning a value to cbWndExtra will cause additional memory to be allocated for each window of the class. If, for example, cbWndExtra is set to 4, every window created using that class will have 4 extra bytes allocated for it. This memory is accessible only by using the GetWindowWord and GetWindowLong functions, and specifying a handle to the window. These values can be set by calling the SetClassWord or SetClassLong functions. As with the class structures, the offset is always specified in bytes.

An example of using window extra bytes would be a text editor that has a variable number of files open at once. The file handle and other file-specific variables can be stored in the window extra bytes of the corresponding text window. This eliminates the requirement to always consume memory for the maximum number of handles or to search a data structure each time a window is opened or closed.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

AllocConsole() Necessary to Get Valid Handles

PSS ID Number: Q89750

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

If a graphical user interface (GUI) application redirects a standard handle, such as stderr or stdout, and then spawns a child process, the output of the child process will not be seen unless the AllocConsole() application programming interface (API) is called before the standard handle is redirected.

If an application spawns a child process without calling AllocConsole() first, the child's console window will appear on the screen and the GUI application will not be able to control this window (for example, it cannot minimize the child window). In addition, users can terminate the child process by choosing Close from the console window's Control (system) menu. This causes users to think that only the window is closed, when in actuality, the entire application is terminated. This can cause the user to lose data in the console window.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseCon

Allowing Only One Application Instance on Win32s

PSS ID Number: Q124134

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, 1.15a, and 1.2

SUMMARY

=======

The entry point for both Windows-based and Win32-based applications is:

```
   int WinMain( hInstance, hPrevInstance, lpszCmdLine, nCmdShow )

   HINSTANCE hInstance;        /* Handle of current instance  */
   HINSTANCE hPrevInstance;    /* Handle of previous instance */
   LPSTR lpszCmdLine;          /* Address of command line     */
   int nCmdShow;               /* Show state of window        */
```

You can allow only one instance of your Windows-based application to run at a time by using hPrevInstance to determine if there is already an existing application instance; then exit the process if there is one. If there is no previous instance, hPrevInstance is NULL.

However, in a Win32-based application, hPrevInstance is always NULL. Therefore, you cannot determine if another instance of your application has been started simply by examining hPrevInstance. This article gives you a method you can use.

MORE INFORMATION

================

Use one of the following four methods to determine if there is an existing application instance on Win32s:

- Synchronize with a named object, such as a file mapping.

  -or-

- Synchronize with a global atom.

  -or-

- Synchronize with a private message.

  -or-

- Use FindWindow() to check for the application.

Using a File Mapping

Using a file mapping works well on any Win32 platform. The global atom is a

cheaper resource, whereas a file mapping will cost a page of memory. A private message is good if you want to inform the first instance that the user attempted to start a second instance, and then let it handle the request -- post a message, become the active application, and so on.

NOTE: You need to clean up before terminating the second instance. FindWindow() doesn't require cleanup, but this method will not work as reliably in a preemptive multitasking environment, such as Windows NT, because you can get in a race condition.

The following code fragment demonstrates how a file mapping can be used to allow only one instance of a Win32-based application. This code should avoid any race conditions. Place this code at the beginning of WinMain().

The code creates a file mapping named MyTestMap using CreateFileMapping(). If MyTestMap already exists, then you know that there is already a running instance of this application. A similar technique would be used with a global atom.

Sample Code

```
   HANDLE hMapping;

   hMapping = CreateFileMapping( (HANDLE) 0xffffffff,
                                 NULL,
                                 PAGE_READONLY,
                                 0,
                                 32,
                                 "MyTestMap" );
   if( hMapping )
   {
      if( GetLastError() == ERROR_ALREADY_EXISTS )
      {
      //
      // Display something that tells the user
      // the app is already running.
      //
         MessageBox( NULL, "Application is running.", "Test", MB_OK );
         ExitProcess(1);
      }
   }
   else
   {
   //
   // Some other error; handle error.
   //
      MessageBox( NULL, "Error creating mapping", "Test", MB_OK );
      ExitProcess(1);
   }
```

Additional reference words: 1.10 1.20
KBCategory: kbprg kbcode
KBSubcategory: W32s

Alternative to PtInRegion() for Hit-Testing

PSS ID Number: Q121960

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

It may be useful to perform hit-testing on an object that is defined by a polygon. To accomplish this, you could call CreatePolygonRgn() to create a region from the polygon, and then call PtInRegion() to determine if the point falls within the region. However, this method can be expensive both in terms of GDI resources, and in terms of speed. If a polygon is complex, CreatePolygonRgn() will often fail due to lack of memory in Windows because regions are in GDI's heap.

The code below provides a better method. Use it to determine if a point lies within a polygon. It is fast and does not use regions. The trick lies in determining the number of times an imaginary line drawn from the point you want to test crosses edges of your polygon. If the line crosses edges an even number of times, the point is outside the polygon. If it crosses an odd number of times it is inside. The line is drawn horizontally from the point to the right.

MORE INFORMATION

WARNING: ANY USE BY YOU OF THE CODE PROVIDED IN THIS ARTICLE IS AT YOUR OWN RISK. Microsoft provides this code "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and/or fitness for a particular purpose. The references below do not constitute a recommendation. You are encouraged to examine any resource to determine whether or not it meets your needs. These books are not recommended over any others.

The following code is based on an algorithm presented in "Algorithms" by Robert Sedgewick, Addison-Wesley, 1988, 2nd ed. ISBN 0201066734. The algorithm is on p.354, in the section "Inclusion in a Polygon" in the chapter "Elementary Geometric Methods." It is also discussed in "Computer Graphics" by Foley, van Dam, Feiner and Hughes, Addison-Wesley, 1990, 2nd ed. ISBN 0201121107, chapter 2, section 1, p.34.

Sample Code

```
#include "windows.h"
#include "limits.h"
BOOL G_PtInPolygon(POINT *rgpts, WORD wnumpts, POINT ptTest,
                   RECT *prbound) ;
BOOL G_PtInPolyRect(POINT *rgpts, WORD wnumpts, POINT ptTest,
                    RECT *prbound) ;
BOOL Intersect(POINT p1, POINT p2, POINT p3, POINT p4) ;
int  CCW(POINT p0, POINT p1, POINT p2) ;

/*********************************************************************

 * FUNCTION:   G_PtInPolygon
 *
 * PURPOSE
 * This routine determines if the point passed is in the polygon. It uses

 * the classical polygon hit-testing algorithm: a horizontal ray starting

 * at the point is extended infinitely rightwards and the number of
 * polygon edges that intersect the ray are counted. If the number is odd,

 * the point is inside the polygon.
 *
 * RETURN VALUE
 * (BOOL) TRUE if the point is inside the polygon, FALSE if not.
 *********************************************************************/

BOOL G_PtInPolygon(POINT *rgpts, WORD wnumpts, POINT ptTest,
                   RECT *prbound)
{
   RECT   r ;
   POINT  *ppt ;
   WORD   i ;
   POINT  pt1, pt2 ;
   WORD   wnumintsct = 0 ;

   if (!G_PtInPolyRect(rgpts,wnumpts,ptTest,prbound))
      return FALSE ;

   pt1 = pt2 = ptTest ;
   pt2.x = r.right + 50 ;

   // Now go through each of the lines in the polygon and see if it
   // intersects
   for (i = 0, ppt = rgpts ; i < wnumpts-1 ; i++, ppt++)
   {
      if (Intersect(ptTest, pt2, *ppt, *(ppt+1)))
         wnumintsct++ ;
   }

   // And the last line
   if (Intersect(ptTest, pt2, *ppt, *rgpts))
      wnumintsct++ ;

   return (wnumintsct&1) ;
```

```
}

/*********************************************************************

 * FUNCTION:   G_PtInPolyRect
 *
 * PURPOSE
 * This routine determines if a point is within the smallest rectangle
 * that encloses a polygon.
 *
 * RETURN VALUE
 * (BOOL) TRUE or FALSE depending on whether the point is in the rect or

 * not.
 *********************************************************************/

BOOL  G_PtInPolyRect(POINT *rgpts, WORD wnumpts, POINT ptTest,
                     RECT *prbound)
{
   RECT r ;
   // If a bounding rect has not been passed in, calculate it
   if (prbound)
      r = *prbound ;
   else
   {
      int   xmin, xmax, ymin, ymax ;
      POINT *ppt ;
      WORD  i ;

      xmin = ymin = INT_MAX ;
      xmax = ymax = -INT_MAX ;

      for (i=0, ppt = rgpts ; i < wnumpts ; i++, ppt++)
      {
         if (ppt->x < xmin)
            xmin = ppt->x ;
         if (ppt->x > xmax)
            xmax = ppt->x ;
         if (ppt->y < ymin)
            ymin = ppt->y ;
         if (ppt->y > ymax)
            ymax = ppt->y ;
      }
      SetRect(&r, xmin, ymin, xmax, ymax) ;
   }
   return (PtInRect(&r,ptTest)) ;
}

/*********************************************************************

 * FUNCTION:   Intersect
 *
 * PURPOSE
 * Given two line segments, determine if they intersect.
 *
```

```
 * RETURN VALUE
 * TRUE if they intersect, FALSE if not.
 ************************************************************************/

BOOL Intersect(POINT p1, POINT p2, POINT p3, POINT p4)
{
   return ((( CCW(p1, p2, p3) * CCW(p1, p2, p4)) <= 0)
        && (( CCW(p3, p4, p1) * CCW(p3, p4, p2)  <= 0) )) ;
}

/************************************************************************

 * FUNCTION:   CCW (CounterClockWise)
 *
 * PURPOSE
 * Determines, given three points, if when travelling from the first to
 * the second to the third, we travel in a counterclockwise direction.
 *
 * RETURN VALUE
 * (int) 1 if the movement is in a counterclockwise direction, -1 if
 * not.
 ************************************************************************/

int CCW(POINT p0, POINT p1, POINT p2)
{
   LONG dx1, dx2 ;
   LONG dy1, dy2 ;

   dx1 = p1.x - p0.x ; dx2 = p2.x - p0.x ;
   dy1 = p1.y - p0.y ; dy2 = p2.y - p0.y ;

   /* This is basically a slope comparison: we don't do divisions because

    * of divide by zero possibilities with pure horizontal and pure
    * vertical lines.
    */
   return ((dx1 * dy2 > dy1 * dx2) ? 1 : -1) ;
}

/*********************************************
 * The above code might be tested as follows:
 *********************************************/
void PASCAL TestProc( HWND hWnd )
{
    POINT rgpts[] = {0,0, 10,0, 10,10, 5,15, 0,10};
    WORD wnumpts = 5;
    POINT ptTest = {3,10};
    RECT prbound = {0, 0, 20, 20};
    BOOL bInside;

    bInside = G_PtInPolygon(rgpts, wnumpts, ptTest, &prbound);

    if (bInside)
       MessageBox(hWnd, "Point is inside!", "Test", MB_OK );
    else
```

```
       MessageBox(hWnd, "Point is outside!", "Test", MB_OK );
}
/* code ends */
```

Additional reference words: 3.00 3.10 3.50 4.00 95 hittest hit-test fails
KBCategory: kbgraphic kbcode
KBSubcategory: GdiMisc

Alternatives to Using GetProcAddress() With LoadLibrary()

PSS ID Number: Q92862

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

When loading a DLL with LoadLibrary(), an alternative to calling GetProcAddress() for each of your DLL entry points is to have the DLL initialization function initialize a global structure or array containing the addresses of these DLL entry points, then call a DLL function from your executable which will return the address of this structure or array to your executable. You can then call your DLL functions via the function pointers in this structure or array.

MORE INFORMATION

The best place to initialize this structure or array of function pointers would be in the DLL_PROCESS_ATTACH code of your DLL's main entry point. The structure or array containing these function pointers must be declared as either a global variable or as dynamically allocated memory (malloc(), GlobalAlloc(), etc.) in your DLL in order for the executable to be able to address this memory properly.

It is also possible, though not as clean, to export the global structure or array of function pointers so that your executable can use the structure or array by name directly in your executable. For more information on how to declare and export global data in a Win32 DLL, please see the following article(s) in the Microsoft Knowledge Base:

   ARTICLE-ID: Q90530
   TITLE   : Exporting Data From a DLL

Be careful not to call these DLL functions via the function pointers after the DLL is unloaded via FreeLibrary(). After FreeLibrary() is called, these function pointer addresses are invalid and calling them will result in an access violation.

This technique of returning pointers to DLL entry points is a supported technique and will work on all hardware platforms that Windows NT supports.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseDll

An Efficient Animation Algorithm

PSS ID Number: Q75431

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

An application that shows an animated image cannot rely solely on Windows graphical device interface (GDI) functions because they will be too slow. Instead, the application must create its own set of bitmaps. This article discusses the process required and provides tips to improve performance and memory use.

This information applies to any type of animation or fast drawing, from painting the game pieces in Reversi to updating the time each second in Clock.

MORE INFORMATION

There are three major steps to this process:

1. Allocate the bitmap.

It is preferable to allocate a single bitmap to store all the different "cels"--the components of the animated scene. The contents of the bitmap should be arranged in a column that is wide enough to hold a single cel; the height is determined by the number of cels. To improve memory usage, the bitmap should be discardable.

For example, given the definitions of the three constants below, the following code allocates the correct size bitmap:

```
X_SIZE = width of the cel
Y_SIZE = height of the cel
NUM_CELS = number of cels in the animated sequence

HBITMAP hbm;

hbm = CreateDiscardableBitmap(hDC, X_SIZE, NUM_CELS * Y_SIZE);
if (!hbm)
    {
    // error - could not allocate bitmap
    }
```

2. Prepare the bitmaps.

To draw into the bitmap, it must be selected into a display context (DC). Allocate a (temporary) compatible DC for this purpose:

```
   if (hTmpDC = CreateCompatibleDC(hDC))
       {
       HBITMAP hOldBm;

       hOldBm = SelectObject(hTmpDC, hbm);
       // and so forth
       }
```

In many cases, all cels will share the same background. Rather than drawing this background several times onto the bitmap, draw it once onto the first cel and copy it to the other cels, as the following code demonstrates:

```
   // GDI calls to draw to hbm from (0, 0) to (X_SIZE, Y_SIZE)

   for (i = 1; i < NUM_CELS; i++)  // Perform the copy
       BitBlt(hTmpDC, 0, i * Y_SIZE, X_SIZE, Y_SIZE, hTmpDC, 0, 0,
                 SRCCOPY);
```

After the background is copied, draw the foreground on each cel, using regular GDI calls (in TRANSPARENT drawing mode). The coordinates for cel "i" in bitmap hbm are:

```
   x_pos: 0 to (X_SIZE - 1)
   y_pos: (i * Y_SIZE) to (((i + 1) * Y_SIZE) - 1)
```

If the cels in the bitmap contain sequential images, animating to the screen is simplified.

To finish this step, release the temporary DC.

```
   SelectObject(hTmpDC, hOldBm);
   DeleteDC(hTmpDC);
```

3. Animate.

A temporary, off-screen DC is required to allow the application to select the bitmap. Note that selecting the object may fail if the bitmap has been discarded. If this has occurred, another bitmap must be allocated (if memory allows) and the bitmap must be initialized (as outlined in step 2, above).

```
   if ((hTmpDC = CreateCompatibleDC(hDC)) != NULL)
       {
       HBITMAP hOldBm;

       if (!(hOldBm = SelectObject(hTmpDC, hbm))
           // must re-allocate bitmap. Note that this MAY FAIL!!!
```

At this point, call the BitBlt() function to copy the various stages of the

animation sequence to the screen. If the cels in the bitmap contain sequential images, a simple loop will do the job nicely, as the following code demonstrates:

```
       for (i = 0; i < NUM_CELS; i++)
           {
           BitBlt (hDC, x_pos, y_pos, X_SIZE, Y_SIZE, hTmpDC, 0,
                       i * Y_SIZE, SRCCOPY);

           // Some form of delay goes here. A real-time wait, based on
           // clock ticks, is recommended.
           }
```

When the drawing is done, delete the temporary DC:

```
   SelectObject(hTmpDC, hOldBm);
   DeleteDC(hTmpDC);
```

It is important to cancel the selection of the bitmap between passes through the for loop. This allows the bitmap to be discarded if the system runs low on memory.

Additional reference words: 3.00 3.10 3.50 4.00 95 animation
KBCategory: kbgraphic
KBSubcategory: GdiBmp

Animating Textures in Direct3D Immediate Mode

PSS ID Number: Q153158

The information in this article applies to:

- Microsoft DirectX 2 Software Development Kit (SDK), for Windows 95

SUMMARY

=======

A texture in Direct3D immediate mode is stored in a DirectDraw surface with the DDSCAPS_TEXTURE flag set in the dwCaps field of the ddsCaps structure. Since the textures are stored in DirectDraw surfaces, you can alter the bits in these surfaces to perform texture animation in your Direct3D applications. To make changes to the pixels in a texture, you would lock down the DirectDraw surface associated with that texture with Lock() and then you would make the changes necessary. You would then load the revised texture on the video card when 3D hardware is used with IDirect3DTexture::Load(). The next time the execute buffer is executed with IDirect3DDevice::Execute(), the revised texture will be used when rendering.

MORE INFORMATION

================

It is important to create a source texture DirectDraw surface in system memory and a destination DirectDraw surface (in video memory, if 3D hardware is present). The source surface should be created with the DDSCAPS_TEXTURE flag set and it should be created in system memory. The destination texture surface should be created with both the DDSCAPS_TEXTURE and DDSCAPS_ALLOCONLOAD flags set. The destination surface will created initially empty. You should call QueryInterface() on both the source and destination textures to get the IDirect3DTexture interface for the texture surfaces. After you have loaded the source texture surface with the bitmap data desired, you should call the destination texture's IDirect3DTexture::Load() method, specifying the source texture, to load the texture onto the destination surface. You can now call the destination texture's IDirect3DTexture::GetHandle() method to obtain a Direct3D texture handle to be used in the execute buffer. When you are done with your texture animation and these textures and surface are no longer required, you should call Release() on the source and destination textures as well as the source and destination surfaces.

The following five steps describe how to implement texture animation in a rendering loop for a Direct3D immediate mode application:

1. Call the destination texture's Unload() method to unload the current texture.

2. Lock down the source texture surface with its Lock() method. Place the new texture data in the surface. Unlock the source texture surface with its Unlock() method.

3. Call the destination texture's Load() method to load the new texture.
   Make sure to specify the source texture as the texture to be loaded.

4. Call the destination texture's GetHandle() method to obtain the
   texture handle for the destination Direct3DTexture object. This handle
   is used in all Direct3D API calls where a texture is to be referenced.

5. Execute your current execute buffer. The revised texture will be used
   for rendering.

Sample Code

Here is a code example to implement the steps above:

```
   lpTexture->lpVtbl->Unload(lpTexture);

   ddsd.dwSize = sizeof(ddsd);
   if (lpSrcTextureSurf->lpVtbl->Lock(lpSrcTextureSurf, NULL,
                              &ddsd, DDLOCK_WAIT, NULL) == DD_OK)
    {
    lpDst = (BYTE *)ddsd.lpSurface;

       // Modify the surface with the new texture bitmap.

    lpSrcTextureSurf->lpVtbl->Unlock(lpSrcTextureSurf,NULL);
    }

   lpTexture->lpVtbl->Load(lpTexture, lpSrcTexture);
   lpTexture->lpVtbl->GetHandle(lpTexture,
                              lpDev, &TextureHandle);

   if (lpDev->lpVtbl->BeginScene(lpDev) != D3D_OK)
       return FALSE;
   if (lpDev->lpVtbl->Execute(lpDev, lpD3DExBuf,
                              lpView, D3DEXECUTE_UNCLIPPED) != D3D_OK)
       return FALSE;
   if (lpDev->lpVtbl->EndScene(lpDev) != D3D_OK)
       return FALSE;
```

Additional reference words: 4.00
KBCategory: kbgraphic kbhowto
KBSubcategory: Direct3D

AppInit_DLLs Registry Value and Windows 95

PSS ID Number: Q134655

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

Windows 95 does not support the loading of a DLL into a process' address space through the use of the AppInit_DLLs registry value. In Windows NT, for every process executed, Windows NT loads the DLLs listed in the AppInit_DLLs registry value into the process' address space. For similar functionality in Windows 95, you can implement a system-wide hook. This article shows by example how to do it.

MORE INFORMATION
==========

To implement a system-wide hook, you must ensure that the hooked function (callback function) exists in a DLL. Then when the this function is called the operating system maps the hooked DLL into the target application's address space. The actual hooked function then operates as part of the target application's process.

There are essentially two steps involved in creating a system-wide hook:

1. Create a DLL with an exported function that is used as the hooking function. In the sample function that follows, the callback function is modeled after a callback function required to implement a WH_KEYBOARD system-wide hook:

```
// Trap keyboard messages
__declspec(dllexport) LRESULT CALLBACK HookFunction(
                int code,
                WPARAM wParam,
                LPARAM lParam)
{
   char szVCode[50];

   //display the virtual key code trapped
   sprintf(szVCode, "Virtual Key code: %lx", wParam);
   MessageBox(NULL, szVCode,"Key stroke", MB_OK);
          :
          :
}
```

   The associated .def file for this DLL might resemble this:

```
LIBRARY      HOOK
```

```
   EXPORTS
      HookFunction
```

2. Install the system-wide hook. To install the hook, the DLL must be loaded, the hook function's address retrieved, and SetWindowsHookEx called with the function's address. Here's an example:

```
   // add system-wide hook
   hHookDll = LoadLibrary("hook");
   hHookProc = (HOOKPROC) GetProcAddress(hHookDll, "HookFunction");

   // Install keyboard hook to trap all keyboard messages
   hSystemHook =  SetWindowsHookEx(WH_KEYBOARD,hHookProc,hHookDll,0);
```

   Once the application has finished with the system-wide hook, be sure to undo the hooking process as follows:

```
   // Remove the hook and unload the DLL used for the hooking process
   UnhookWindowsHookEx(hSystemHook);
   FreeLibrary(hHookDll);
```

Additional reference words: 4.00 95
KBCategory: kbprg kbcode
KBSubcategory: BseMisc

Application Can Allocate Memory with DdeCreateDataHandle

PSS ID Number: Q85680

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

=======

An application can use the DdeCreateDataHandle function to create a handle to a block of data. The application can use the handle to pass the data to another application in a dynamic data exchange (DDE) conversation using the Dynamic Data Exchange Management Libraries (DDEML). All DDEML functions refer to blocks of memory using data handles.

An application can allocate memory and manually create a data handle associated with the memory (using method 1 below), or automatically by using the DdeCreateDataHandle function (method 2 below).

Method 1

----

1. Obtain a block of memory using the GlobalAlloc or LocalAlloc function or by declaring a variable in your application.

2. Fill the block of memory with the desired data.

3. Call the DdeCreateDataHandle function to create a data handle associated with the block of memory.

Method 2

----

1. Call the DdeCreateDataHandle function with the lpvSrcBuf parameter set to NULL, the cbInitData parameter set to zero, and the offSrcBuf parameter set to the number of bytes of memory required.

2. To retrieve a handle to the memory block, specify the data handle returned by DdeCreateDataHandle as the hData parameter of the DdeAccessData function. This operation is similar to calling the GlobalLock function on a handle returned from GlobalAlloc.

3. Use the pointer to fill the memory block with data.

4. Call DdeUnaccessData to unaccess the object. This operation is similar to calling the GlobalUnlock function on a handle returned

from GlobalAlloc.

The following code fragment demonstrates method 2:

```
// Retrieve the length of the data to be stored
cbLen = lstrlen("This is a test") + 1;

// Create the data handle and allocate the memory
hData = DdeCreateDataHandle(idInst, NULL, 0, cbLen,
                            hszItem, wFmt, 0);

// Access the data handle
lpstrData = (LPSTR)DdeAccessData(hData, NULL);

// Fill the block of memory
lstrcpy(lpstrData, "This is a test");

// Unaccess the data handle
DdeUnaccessData(hData);
```

When an application obtains a data handle from DdeCreateDataHandle, the application should next call DdeAccessData with the handle. If a data handle is first specified as a parameter to a DDEML function other than DdeAccessData, when the application later calls DdeAccessData, the application receives only read access to the associated memory block.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Application Exception Error Codes

PSS ID Number: Q101774

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

Many exception errors are not processed by applications. The most common exception error is EXCEPTION_ACCESS_VIOLATION (c0000005). It occurs when a pointer is dereferenced and the pointer points to inaccessible memory or a write operation is attempted on read-only memory. If an application does not trap an exception, the Win32 module, UnhandledExceptionFilter, will do one of the following: display a message box, invoke Dr. Watson, or attach your application to a debugger.

The following are standard exception errors:

```
   EXCEPTION_ACCESS_VIOLATION
   EXCEPTION_ARRAY_BOUNDS_EXCEEDED
   EXCEPTION_BREAKPOINT
   EXCEPTION_DATATYPE_MISALIGNMENT
   EXCEPTION_FLT_DENORMAL_OPERAND
   EXCEPTION_FLT_DIVIDE_BY_ZERO
   EXCEPTION_FLT_INEXACT_RESULT
   EXCEPTION_FLT_INVALID_OPERATION
   EXCEPTION_FLT_OVERFLOW
   EXCEPTION_FLT_STACK_CHECK
   EXCEPTION_FLT_UNDERFLOW
   EXCEPTION_ILLEGAL_INSTRUCTION
   EXCEPTION_IN_PAGE_ERROR
   EXCEPTION_INT_DIVIDE_BY_ZERO
   EXCEPTION_INT_OVERFLOW
   EXCEPTION_INVALID_DISPOSITION
   EXCEPTION_NONCONTINUABLE_EXCEPTION
   EXCEPTION_PRIV_INSTRUCTION
   EXCEPTION_SINGLE_STEP
   EXCEPTION_STACK_OVERFLOW
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseExcept

Application Version Marking in Windows 95

PSS ID Number: Q125705

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

Applications designed for Windows 95, whether 16- or 32-bit, should be marked for Windows version 4.0 so they receive the full benefit of new user interface features in Windows 95. Applications marked as being designed for earlier versions of Windows will display behavior consistent with the Windows version 3.1 user interface, which is not always identical to Windows 95 behavior.

Executables marked for Windows version 4.0 will load on Windows 95, Windows NT version 3.5, Win32s version 1.15, and later versions, but not on earlier versions.

NOTE: This article deals solely with marking executable files as compatible with a particular Windows version. This is different from the version resources (VS_VERSION_INFO) that may be contained in an executable.

MORE INFORMATION

The Microsoft Visual C++ version 2.1 linker defaults to marking executables for Windows version 4.0 while the version 2.0 linker defaults to 3.1. The Books Online section "Appendix B: Link Reference" incorrectly states that versions 2.1 and 2.2 default to a subsystem of 3.10. To override the default, the Microsoft Visual C++ linkers accept the following syntax for the /SUBSYSTEM option:

```
/SUBSYSTEM:WINDOWS,4.0
```

In the development environment, you can change the /SUBSYSTEM option by going to the Project menu, selecting Settings, selecting either Win32 Debug or Win32 Release, choosing the Link tab, and editing the Project Options. To explicitly set the subsystem to the version prior to 4.0, specify a subsystem of 3.10. The trailing 0 in 3.10 is required in this case.

You may need to perform a full link for this to take effect, but subsequent incremental linking with this switch will work correctly.

To mark a 16-bit executable as Windows version 4.0 compatible, use the 16-bit resource compiler (RC.EXE) from the Windows 95 SDK to bind the resources into the executable file. By default, this version of RC marks the executable for version 4.0, but this can be overridden by using the -30 or -31 switch.

An application will display several behavioral differences depending on which Windows version the application is compatible with:

1. All standard control windows owned by a version 4.0 application are drawn with a chiseled 3D look. To obtain the same effect for dialogs owned by a version 3.1 application, use the DS_3DLOOK dialog style. This style is ignored if the application is run on Windows platforms other than Windows 95. The dialog style DS_3DLOOK is not defined in the standard WINRES.H. Therefore, to use this style, you must define it as

   ```
   #define DS_3DLOOK 0x0004L
   ```

   and or DS_3DLOOK into the STYLE line of each dialog in the project's .RC file. NOTE: Editing a dialog template using the resource editor removes this DS_3DLOOK style bit.

   NOTE: This symbol will be defined in a future release of Visual C++. At that time, you will get the error "symbol redefined". At that point, you can remove your definition of DS_3DLOOK.

2. Thunks created using the Windows 95 SDK Thunk Compiler will not work unless the 16-bit thunk DLL is marked for version 4.0.

3. Windows version 3.1 allowed 16-bit applications to share GDI resources such as font or bitmap handles. For backwards compatibility, Windows 95 does not clean up objects left undeleted by a 16-bit version 3.x application when that application terminates because these objects may be in use by another application. Instead, such objects remain valid as long as there are any 16-bit applications running. When all 16-bit applications are closed, these objects are freed.

   On the other hand, is is assumed that 16-bit version 4.0 applications follow the Windows 95 guidelines, and do not share objects. Thus, all objects owned by a 16-bit version 4.0-based application are freed when the application terminates.

   Win32-based applications cannot freely share GDI objects, so all owned objects are freed when a Win32-based application terminates regardless of the version of the application.

4. New window messages, such as WM_STYLECHANGED, are only sent to windows owned by a version 4.0 application.

5. Printer DCs are reset during StartPage() in applications marked with version 4.0

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrMisc

Assigning Mnemonics to Owner-Draw Push Buttons

PSS ID Number: Q67716

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

An application that uses owner-draw push buttons is always responsible for the appearance of the buttons. It might seem that in doing so, the ability to assign a mnemonic character to an owner-draw button is lost because text containing the mnemonic may not be displayed.

Fortunately, this is not the case. If an owner-draw button should be activated by ALT+X, place "&X" into the button text. NOTE: You have to use DrawText() to get the & character to underline the next character. Using TextOut() will not cause the & character to underline the next character in the string.

When the ALT key is pressed in combination with any character, Windows examines the text of each control to determine which control, if any, uses that particular mnemonic. With an owner-draw button, the text exists, but may not necessarily be used to paint the button.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Associating Data with a List Box Entry

PSS ID Number: Q74345

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

In the Microsoft Windows graphical environment, an application can use the LB_SETITEMDATA and LB_GETITEMDATA messages to associate additional information with each entry in a list box. These messages enable an application to associate an arbitrary LONG value with each entry and to retrieve that value. This article documents how an application uses these messages.

MORE INFORMATION
================

In this example, the application will associate a 64-byte block of data with each list box entry. This is accomplished by allocating a global memory block and using the LB_SETITEMDATA message to associate the handle of the memory block with the appropriate list box item.

During list box initialization, the following code is executed for each list box item:

```
   if ((hLBData = GlobalAlloc(GMEM_MOVEABLE, 64)))
      {
      if ((lpLBData = GlobalLock(hLBData)))
         {
         // Store data in 64-byte block.

         GlobalUnlock(hLBData);
         }
      }
// NOTE: The MAKELONG is not needed on 32-bit platforms.
   SendMessage(hListBox, LB_SETITEMDATA, nIndex, MAKELONG(hLBData, 0));
```

To retrieve the information associated with a list box entry, the following code can be used:

```
// NOTE: The return from LB_GETITEMDATA is a long on 32-bit platforms.
   if ((hLBData = LOWORD(SendMessage(hListBox, LB_GETITEMDATA,
              nIndex, 0L))))
      {
      if ((lpLBData = GlobalLock(hLBData)))
```

```
         {
         // Access or manipulate the data or both.

         GlobalUnlock(hLBData);
         }
      }
```

Before the application terminates, it must free the memory associated with each list box item. The following code frees the memory associated with one list box item:

```
   if ((hLBData = LOWORD(SendMessage(hListBox, LB_GETITEMDATA,
         nIndex, 0L))))
      GlobalFree(hLBData);
```

These techniques can be used to associated data with an entry in a combo box by substituting the CB_SETITEMDATA and CB_GETITEMDATA messages.

Additional reference words: 3.00 3.10 3.50 4.00 95 combobox listbox
KBCategory: kbui
KBSubcategory: UsrCtl

Authoring Windows Help Files for Performance

PSS ID Number: Q74937

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

The Windows Help Compiler allows an application developer to create hypertext documentation, richly annotated with color and graphics. This article discusses methods to author help files to achieve maximum performance when the file is used. These comments apply to Windows Help versions 3.06 and later.

This information is subject to change in future versions of the Help Compiler and of the Help application as new technology is incorporated into these products.

There are four major suggestions:

1. Use OPTCDROM=1 for all files destined for CD-ROM (compact-disk read-only memory), and potentially on files where up to an additional 10K of size is not significant.

2. Use bitmaps placed with data for small bitmaps that are referenced only once or infrequently in the help file.

3. Use bitmaps not placed with data for large bitmaps, all bitmaps referenced frequently, or bitmaps referenced by two or more topics that generally will be viewed in succession.

4. Use segmented hypergraphics to generate graphics with multiple hot spots, rather than several bitmaps positioned next to each other.

Coincidentally, suggestions 2, 3, and 4 are also generally space-saving techniques as well.

MORE INFORMATION

1. Use OPTCDROM=1 for files destined for CD-ROM.

   When OPTCDROM is placed in the [OPTIONS] section of the .HPJ file, the topic information in the help file is aligned on 2K boundaries. This option is aimed at maximizing performance on CD-ROM drives, where reading aligned information can be significantly faster.

   Estimates indicate that sequential reads from CD-ROM can be up to twice as fast when aligned. While reads are not always sequential, a high percentage can be, depending on how the help file is

authored. Minor improvements have also been noted on magnetic media.

This option can cause up to 10K of additional file space to be used.

2. Store small bitmaps with data.

   Placing bitmaps with data keeps the graphical and textual information in the same location in the help file. This avoids reading from different locations on disk to display a topic. Seeks to different locations are exceptionally time consuming on CD-ROM, and can be time consuming on magnetic media. Bitmaps with data also help maximize the effects of the OPTCDROM option.

   Up to 12K of compressed topic text is buffered in memory. Since bitmaps are kept with the topic text, they take advantage of this buffering. Thus, topics under 12K in compressed size generally do not need to be reread when the window is resized or redrawn.

   Note that large bitmaps kept with data may cause performance to become worse. Topics may be laid out twice as part of scroll bar removal. Topics over 12K in size will be read, laid out, and then reread as they are laid out a second time before display. Bitmaps are often the cause of exceeding the 12K size. Bitmaps NOT kept with data are buffered elsewhere (see #3 below), and if the remaining textual data is less than 12K, the topic will be read from disk only once.

   Bitmaps with data cost space only when the same bitmap is used more than once in the help file. If a bitmap is used frequently, not placing it "with data" may be more appropriate, unless the performance benefit is determined to outweigh the size hit.

3. Store large and frequently accessed bitmaps separately.

   The 50 most recently accessed bitmaps not stored with data are cached in memory. This means that the bitmap may have to be read from disk only once if two successively displayed help topics reference it. Unlike bitmaps stored with data, these cached bitmaps only have to be reread if they are bumped out of the cache by the subsequent display of 50 more bitmaps, or by low memory conditions.

   Cached bitmaps not stored with data cost some speed when they are typically displayed by the user only once in a session. Since they are stored in a different portion of the help file from the topic text, an additional seek is required to locate them. This cost, if incurred, is generally negligible on disk, and high on CD-ROM. Note, however, that this cost MAY be less than the cost of reading the topic twice, if the topic is laid out twice as a result of not needing scroll bars, and is larger than 12K.

4. Use segmented hypergraphics.

   Segmented hypergraphics is the term used to describe the ability to

take a single bitmap, and define several regions that are hot spots. Hot spots can overlap, and can each perform different actions.

The primary benefit of using segmented hypergraphics is that a single bitmap can be used. Previous techniques utilizing several bitmaps carefully placed in the topic text to generate a single image have the drawback of requiring several bitmap-locating operations at display time, which can translate to several disk seeks and reads. On CD-ROM, especially, this can be quite significant.

The only cost involved in using segmented hypergraphics is that the segmented hypergraphics editor must be used to define the hot spots within the bitmaps.

In summary, there are a few simple authoring techniques that can improve performance of help files. While especially significant for CD-ROM hosted help files, they are of benefit to disk-based help files.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsHlp

Availability of Multimedia Timers

PSS ID Number: Q140104

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, 4.0
- Microsoft Win32s version 1.2

SUMMARY

=======

The number of multimedia timers your application can create at one time can vary based on the operating system it is running on and whether the application is 16-bit or 32-bit.

MORE INFORMATION

================

The following table gives an indication of what you can expect:

Operating System	16-bit app	32-bit app
Windows 3.11	8	0 (Not supported on Win32s)
Windows 95	32	32
Windows NT 3.51	16	16 per process

Note that a multimedia timer is created by using the timeSetEvent() API. With the exception of a 32-bit application running under Windows NT, these numbers represent the total number of available timers in the entire system. With this in mind, the number of multimedia timers your application can allocate at one time could be less then the amount shown in the table. Win32s does not support multimedia callbacks, so the timeSetEvent() API is not available under Win32s.

Additional reference words: limit

KBCategory: kbmm kbprg

KBSubcategory: MMTimer

Availability of Microsoft Network SDKs

PSS ID Number: Q115604

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK), version 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0
- Windows for Workgroups SDK, version 3.11
- Microsoft LAN Manager SDK

SUMMARY
=======

The following SDKs are available for programmers writing network applications on Microsoft platforms:

Microsoft Windows for Workgroups SDK

This is available on the April 1994 (and later) Microsoft Developer Network (MSDN) Level 2 CDs.

The SDK, documentation, debug kernel, and other files for network programmers on Windows for Workgroups, version 3.11 are available on the April 1994 MSDN. This SDK includes APIs for the network extensions to Windows for Workgroups, version 3.11.

Microsoft LAN Manager SDK

The documentation is available in two books:

- "Microsoft LAN Manager, A Programmer's Guide," Ralph Ryan, Microsoft Press, ISBN 1-55615-166-7.

- "Microsoft LAN Manager Programmer's Reference," Microsoft Corporation, Microsoft Press, ISBN 1-55615-313-9.

This SDK includes APIs for writing distributed applications and administration programs for Microsoft LAN Manager.

LAN Manager APIs for Windows NT

The file DOC\SDK\MISC\LMAPI.HLP on the CD-ROM describes the ported LAN Manager APIs. Windows NT supports 32-bit equivalents of most of the LAN Manager APIs.

The LAN Manager APIs are included in the header file LMACCESS.H and in the import library NETAPI32.LIB. These APIs are described in the Win32 API help file.

Windows Sockets APIs

The Windows Sockets APIs are available through the SDKs. The file WINSOCK.HLP gives the details.

The files needed to support Windows Sockets APIs (conforming to the Windows Sockets specifications) for Microsoft LAN Manager may be obtained on the Internet (FTP.MICROSOFT.COM, ADVSYS\LANMAN\SUP-ED\WINSOCK).

The Windows Sockets specifications, libraries, header files and samples may also be obtained from the Internet in the ADVSYS\WINSOCK directory.

Microsoft Remote Procedure Call (RPC)

The support for RPC is included in the Win32 SDKs. The Win32 API help file includes the RPC APIs. The RPC.HLP file gives the details of RPC support.

RPC support files for MS-DOS and Microsoft Windows may be obtained from the Win32SDK CD-ROM. The directory MSTOOLS\RPC_DOS has the required files.

NOTE: Microsoft also provides other kinds of network APIs. For example, the NetBIOSCall() API provides a mechanism to write NetBIOS applications on Microsoft Windows; on Windows NT this is accomplished with the Netbios() API. Information on these other APIs may be obtained from the Windows and Win32 SDK documentation.

Additional reference words: 3.10 3.50 4.00 95 3.11 msdnl2
KBCategory: kbref
KBSubcategory: NtwkMisc

AVI Video Authoring Tips & Compression Options Dialog Box

PSS ID Number: Q139826

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Video for Windows Development Kit (DK) version 1.1

SUMMARY

This article provides tips you can use to help you compress AVI (audio-video interleaved) files. This article should help to clarify the information in the documentation. This article also provides hints concerning the settings for Interleave, Compression Quality, Key Frame Every, and Data Rate - all settings available in the Video Compression Options dialog box.

MORE INFORMATION

Authoring/Compression Tips

- Don't capture on low or medium end equipment.
- Avoid noise. Noise reduces the quality of the image and can effect the compression.
- Use a low-pass filter such as the LOWPASS sample included with Visual C++ version 2.x and with the Win32 SDK to get rid of some of the audio noise.
- Avoid compressing the movie more than once. Edit your uncompressed bits of video together before doing any compression, and then compress the final product. Editing always works faster with uncompressed video.

Video Dimensions and Frame Rate

When authoring, consider that most display cards support up to 320x240x15 frames per second (fps). Larger images or faster frame rates increase the importance of testing on multiple machines with multiple configurations (CD-ROM drive speeds, video boards, display monitors, and so on).

Full Screen Playback

If your movie is 320x240 or less, you should get excellent full-screen playback that can handle 15 fps regardless of the video card. If your movie is bigger, it will play full screen, but performance may be poor. Test across multiple configurations to help identify the impact of stretching on various displays.

Video Compression Options

The following options are presented in the Compression Options dialog box. To gain access to this dialog box, on the File menu, click Save Options in SDK samples such as Aviedit and Aviview. Similar options are available in the Videdit utility from Video for Windows by clicking Save As on the File menu.

Interleave:

The interleave option places audio data physically between video frame data in the AVI file to ensure the best performance and synchronization. For example, with a 1:1 interleave setting, each frame of a 15 fps video would have 1/15 seconds of audio data. This setting does not matter when editing the file. When saving for playback, interleave every frame for best playback performance. Interleaving multiple audio streams is not possible using this option.

Compression Quality:

Use the highest quality number to get the best quality, but use the entire allowable data rate. A lower quality may take up less than the requested data rate.

Key Frame Every:

Normally, use the default key frame value for a particular codec. Fewer key frames could give a little better picture quality, but if your system can't keep up on playback, you'll stall for a longer time, and you won't degrade as gracefully. More key frames means fewer frames skipped when playback can't keep up.

Data Rate:

The minimum data rate that can be achieved on most computers is 100K/sec for single spin CD-ROM drives, and 225K/sec for double spin CD-ROM drives. Most computers can do more (150K and 300K for single and double speed respectively).

REFERENCES

The AVIMakeCompressedStream(), AVISave(), and AVISaveOptions() functions use the AVICOMPRESSOPTIONS structure to specify some of the compression options discussed in this article. The ICCompressorChoose() function uses the COMPVARS structure for a similar purpose. See the documentation for the products listed at the beginning of this article for more information.

Additional reference words: 3.10 4.00 3.50 dwKeyFrameEvery dwQuality dwBytesPerSecond dwInterleaveEvery lKey lDataRate lQ fullscreen authoring author save
KBCategory: kbmm kbprg kbdocerr

KBSubcategory: MMVideo

Avoid Calling SendMessage() Inside a Hook Filter Function

PSS ID Number: Q74857

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

A hook filter function should not call SendMessage() to pass intertask messages because this behavior can create a deadlock condition in Windows. If a hook filter function is called as a result of an intertask SendMessage(), and if the hook function then yields control with an intertask SendMessage(), a message deadlock condition may occur. For this reason, hook filters should use PostMessage() rather than SendMessage() to pass messages to other applications.

NOTE: A hook filter can use SendMessage() to pass a message to the current task because this will not yield the control.

Section 1.1.5 of the "Microsoft Windows Software Development Kit Reference Volume 1" from the Windows SDK version 3.0 documentation has more information on message deadlocks.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrHks

Background Colors Affect BitBlt() from Mono to Color

PSS ID Number: Q41464

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

When using BitBlt() to convert a monochrome bitmap to a color bitmap, GDI transforms all white bits (1) to the background color of the destination device context (DC). GDI transforms all black bits (0) to the text (or foreground) color of the destination DC.

When using BitBlt() to convert a color bitmap to a monochrome bitmap, GDI sets to white (1) all pixels that match the background color of the source DC. All other bits are set to black (0).

These features are mentioned in the BitBlt() documentation.
manual.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiBmp

Background, Foreground, and System Palette Management

PSS ID Number: Q72386

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

On a device that supports the Microsoft Windows palette management APIs, an application can create a logical palette, select the palette into a device context (DC), and realize the palette. Realizing a logical palette maps its colors to the system (hardware) color palette. The GetDeviceCaps() API is available to inform an application whether the device is capable of supporting palette management functions and, if so, the size of its system palette. This article discusses the different types of logical palettes and the effect of each on the system palette when a logical palette is realized.

MORE INFORMATION

When a logical palette is selected into a DC, it can be selected as either a foreground or a background palette. Setting the bForceBackground parameter of the SelectPalette() API to TRUE selects the palette as a background palette. If this parameter is FALSE, the palette can be selected as a foreground palette. A palette will be selected as a foreground palette only if the DC into which the palette is selected is one of the five cached DCs managed by the GetDC() API and the DC is retrieved on behalf of the active window. If the DC is returned by CreateDC() or CreateCompatibleDC() or if the window is not the active window, the palette will be forced into the background.

The status as a foreground or a background palette affects how the colors in the logical palette are mapped into the system palette when the logical palette is realized.

When a foreground palette is realized, every entry in the system palette that can be modified by an application is accessible to the logical palette. Logical palette entries are mapped into the system palette starting at the first available entry. Because a logical palette entry that exactly matches a reserved system palette entry is mapped to that system entry, it does not consume a separate palette slot. If the logical palette has more entries than available slots in the system palette, the available slots are filled, in order, from the logical palette. The remaining logical palette entries are mapped to the closest colors already present in the

system palette. There is one exception to this rule: if a logical palette entry is marked with the PC_RESERVED flag, no colors will be mapped to that entry. If all available system palette entries are reserved, additional colors will not be mapped to any entry and will be displayed as black on the screen.

A palette entry marked as PC_NOCOLLAPSE will always take a separate slot if available, just as for PC_RESERVED. Unlike a PC_RESERVED color, if no slots are available, it will map to the nearest color, and other colors may map onto it.

The first available entry in the system palette is the first palette entry not marked as used. For example, assume a device with 256 palette entries, 20 of which are reserved for the system. An application realizes a palette of 36 colors on this device; therefore, the first 36 entries are marked used. Another application realizes a 100-entry palette; therefore, the next 100 entries are marked used. If a third application receives the input focus and realizes a foreground palette with 236 entries, Windows maps the first 100 colors into the remainder of the system palette. Each of the remaining 136 colors of the logical palette is mapped into the closest color available in the system palette.

When a background palette is realized, any empty positions in the system palette are filled. Any colors that remain are mapped to the closest color in the system palette. A background palette entry cannot overlay a foreground entry in the system palette; however, a foreground palette entry can overlay a background entry in the system palette.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiPal

BeginPaint() Invalid Rectangle in Client Coordinates

PSS ID Number: Q19963

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The BeginPaint() function returns a pointer to a PAINTSTRUCT data structure through its second parameter. The rcPaint field of this structure specifies the update rectangle in client-area coordinates (relative to the upper-left corner of the window client area).

This update rectangle also serves as the clipping area for painting in the window, unless the invalid area of the window is expanded using the InvalidateRect() function.

Additional reference words: 3.00 3.10 3.5 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrPnt

Binary and Source Compatibility Under Windows NT

PSS ID Number: Q93213

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY

=======

There are currently three hardware platforms for which Win32-based applications can be written; they are Intel (x86), MIPS, and DEC Alpha. Binary compatibility across these hardware platforms is not a viable alternative. Therefore, Windows NT offers source compatibility. This means that developers may create versions of their Win32 applications for each CPU by recompiling.

For example, suppose that you have written a Win32-based application in C and have used a 32-bit compiler that targets the Intel platform. In order to produce a Win32-based application that runs on DEC Alpha, you would purchase a 32-bit compiler that targets Alpha and recompile your code.

MORE INFORMATION

================

It is important not to confuse source compatibility across hardware platforms with binary compatibility across application execution environments. Windows NT and Win32s do support binary compatibility between different application execution environments. For example, the typical 16-bit Windows-based application can be run without modification on any Windows NT machine. On the MIPS and Alpha platforms, this is achieved through emulation. In addition, it is possible to design an x86 Win32-based application so that it runs on Windows 3.1. This is achieved through Win32s.

Additional reference words: 3.10 3.50

KBCategory: kbprg

KBSubcategory: SubSys

Binary Compatibility Basics

PSS ID Number: Q95162

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY
=======

Because of differing instruction sets between processors, object code is not binary-compatible across platforms. To enable object code compiled for processor X to run on processor Y, a virtual machine must be created on processor Y to emulate processor X's instructions, which results in a considerable performance hit. For this reason, Windows NT offers a strong source compatibility, which makes it a relatively simple matter to compile the same code into native object code for each platform.

The Hardware Abstraction Layer (HAL), smooths over differences between hardware implementations. All access to the hardware from the operating system (OS) goes through the HAL, which makes changing of both the hardware and the OS itself much simpler. The HAL, however, does not emulate the instruction set of the platform; a common misconception.

MORE INFORMATION
================

Source code is compiled into executable or object code for the instruction set of a specific processor or family (x86, 680x0, R3000/4000, and so forth). This code runs natively only on that type of processor. Remember, this reduces to 1's and 0's, hence binary, so even if two processors have a large overlap in the available instructions (and they do; MOVs, XORs, or whatever at the assembly language level), what is actually executed is not even relatively "high level" human-readable assembly code but merely a series of bytes that by convention/definition only are assigned the meanings that are human-readable at the assembly language level.

For example, probably every microprocessor has an OR instruction. On the Intel x86, the OR instruction may be represented in the instruction stream by the bytes "09 [effective address]" (see page 456 of Microsoft's 80386/80486 Programming Guide, 2nd Ed.). On the R4000, however, it's nearly guaranteed to be something different.

Thus, if you want to run an executable compiled for Intel on a MIPS chip, you must run a program that behaves as an Intel instruction interpreter (similar to a Basic interpreter, but much more complex). Such a program is called an emulator, and will scan through the Intel object code and then in turn execute equivalent commands in MIPS form on the processor. But the emulator must do much more than that; it must also create a "virtual machine," a complete software execution environment that behaves similar to the original hardware/software environment that the executable was originally compiled for.

Note that even in an ideal case, every processor X instruction requires about 5-20 instructions on processor Y. The object code interpreter must examine the next instruction/byte, compare its value to known instruction values via some kind of table (depending on the implementation), and then execute the equivalent native instruction. There is room for optimization but it will never be very fast (relative to native code).

Therefore, run non-native object code only if you absolutely must. Below is a binary compatibility table for Windows NT:

System	Binary-Compatible on NT?
Win16 apps	Yes (via Insignia's x86 emulation code)
Win32 apps	No (must re-compile)
POSIX apps	No (POSIX is a source-code standard)(1)
OS/2 1.x apps	No (Don't run on non-x86 machines at all)(2)

Notes: (1) POSIX 1003.1 is a C-language source-level standard for basic operating system services. POSIX applications don't need to be binary-compatible even on the same instruction set! That is, a POSIX application compiled and linked for SCO on x86 will NOT run on x86 Windows NT POSIX or x86 Sun/Interactive POSIX. (2) OS/2 1.x support is technically feasible but would have required more work on the non-Intel platforms, and was not considered a high priority.

The Hardware Abstraction Layer (HAL)

A common misconception is that the HAL should allow binary compatibility. The Windows NT HAL has absolutely nothing to do with the issue discussed above; that is, running code complied for processor X on processor Y. Nor is the HAL akin to a "virtual machine" implementation; it does not simulate anything. Rather, the HAL is simply an example of a good modular software design that deals with the issue of differences in hardware design between machines with the same instruction set (or between instruction sets).

The HAL provides a set of services to the rest of the Windows NT executive that abstracts and "hides" the differences between low- level hardware-software interfaces, such as with DMA controllers, programmable interrupt controllers, clocks and timers, and so forth. In a typical pre-Windows NT operating system, there is lots of code embedded throughout the operating system (particularly in device drivers) that is specifically tied to particular implementations of hardware services (a particular PIC, a particular DMA chip, and so forth). If one of these hardware pieces is changed, lots of code scattered throughout the system will break. As a result, the hardware never changed and a typical 486/66 machine today uses the same low-function hardware devices that appear in the IBM AT 286 (if not the IBM PC itself).

In Windows NT, any other part of the OS (including the kernel and all device drivers) that needs to deal with those low-level devices now uses HAL services, and is therefore isolated from changes in the hardware. If you change those hardware pieces you only need to change the HAL. This

results in at least the following two advantages:

- A cleaner, easier to write and debug design for the OS and device drivers.
- The real possibility for innovation and change in the underlying hardware.

But the HAL does not provide an abstract instruction set, or the services necessary for running object code from processor X on processor Y. It is possible to write an such as OS if you are willing to take the huge performance hit, but Windows NT isn't it. And because a well-designed OS such as Windows NT can provide a very complete level of source-code compatibility across instruction sets, and therefore a relatively painless way of getting native-code versions of Win32-based applications, probably no one will be willing to take that performance hit in a mainstream operating system, no matter how fast the hardware.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: SubSys

Broadcasting Messages Using PostMessage() & SendMessage()

PSS ID Number: Q64296

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

When SendMessage() is used to send a broadcast message (hwnd = 0xFFFF or hwnd = -1), the message is sent to all top-level windows. A message broadcast by PostMessage() is only sent to top-level windows that are visible, enabled, and have no owner.

You might observe the effect of the difference when, for example, the top-level window of your application calls DialogBox() to present a modal dialog box. While the modal dialog box exists, its owner (your top-level window) will be disabled. Messages broadcast using PostMessage() will not reach the top-level window because the window is disabled, and will not reach the dialog box because the dialog box has an owner. Messages broadcast using SendMessage() will reach both the top-level window and the dialog.

In Windows 3.1, PostMessage() will broadcast to invisible and disabled windows just like SendMessage() already does.

Both PostMessage() and SendMessage() actually broadcast using the same broadcast procedure. This procedure does some additional screening to make sure that pop-up menus, the task manager window, and icon title windows are insulated from broadcast messages.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMsg

BUG: 16-bit RPC Samples Print Meaningless Error Codes

PSS ID Number: Q139717

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, 4.0

SYMPTOMS
========

Some of the 16-bit RPC samples print meaningless error codes. For example:

```
Runtime reported exception 378535963
```

CAUSE
=====

The 16-bit applications method for generating exceptions are different from those used by 32-bit applications. This is encapsulated to ease programming. Please look in the header file Rpcx86.h.

RpcExceptionCode is defined as _exception_code. However, _excpetion_code is a 16-bit integer, so when this is printed as an unsigned long integer, the result is not meaningful.

RESOLUTION
==========

Any of the following methods should result in meaningful error codes:

- Explicitly cast the value returned by RpcExceptionCode() as an unsigned long. For example:

```
printf("Runtime reported exception %ld\n", (unsigned long)
   RpcExceptionCode());
```

-or-

- Use an unsigned long variable to trap the value of RpcExceptionCode(). For example:

```
unsigned long ulCode;
ulCode = RpcExceptionCode();
printf("Runtime reported exception %ld\n",  ulCode );
```

Note that this method is used by the samples that do not exhibit the problem.

-or-

- Print the value as an unsigned quantity. For example:

```
      printf("Runtime reported exception %u\n", RpcExceptionCode());

      -or-

      printf("Runtime reported exception %ul\n", RpcExceptionCode());
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

This bug is exhibited by the following samples:

```
callback\callc.c
data\dunion\dunionc.c
data\inout\inoutc.c
data\repas\repasc.c
data\xmit\xmitc.c
doctor\doctorc.c
rpcssm\rpcssmc.c
```

REFERENCES

The SDK header file Rpcerr.h that installs with the 16-bit runtimes for RPC.

Additional reference words: 3.10 3.50 3.51 4.00 win16sdk
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkRpc

BUG: 32-Bit .Fon File Causes GP Fault with Standard VGA Driver

PSS ID Number: Q137882

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0

SYMPTOMS

Windows NT and Windows 95 can use 32-bit .fon files in the PE format. These raster font files can be created when using the 32-bit compiler or linker if you create a 32-bit DLL that contains one or more .fnt files in its resources. However, If you run Windows 95 with the "Standard Display Adapter (VGA)(Microsoft)," your application will shut down (perform an illegal operation or produce a general protection (GP) fault) if you try to call TextOut() with a 32-bit .fon raster font selected into the device context you are calling TextOut() on.

RESOLUTION

Create a 16-bit .fon file instead of a 32-bit .fon file if you want your font to function properly on all video drivers. The following article in the Microsoft Knowledge Base describes a technique you can use to create a 16-bit .fon file:

```
ARTICLE_ID: Q76535
TITLE     : SAMPLE: Creating a Custom Raster Font
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

Steps to Reproduce Problem

1. Create a 32-bit .fon raster font file by using the 32-bit compiler.

2. Call CreateFont() to create a logical font for it, and select it into a device context.

3. Try calling TextOut() to draw some text. A GP fault occurs.

Additional reference words: 4.00 TextOut .fnt
KBCategory: kbgraphic kbbuglist

KBSubcategory: GdiFnt

BUG: 32-bit StampResource() Function Won't Work in Windows 95

PSS ID Number: Q133700

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

Setup programs created with the 32-bit Setup Toolkit fail on a call to StampResource() in Windows 95.

CAUSE

StampResource() makes use of a Win32 API called UpdateResource() to write string table resource data into binary (executable) files. UpdateResource() is not implemented in Windows 95, so StampResource() fails to work.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbtool kbbuglist
KBSubcategory: tlsmss

BUG: Bad Characters in 32-bit App on Win32s on Russian Windows

PSS ID Number: Q126865

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5
- Microsoft Win32s versions 1.2 and 1.25

SYMPTOMS

When compiling, linking, and running a Russian application which was built using Microsoft Visual C++ version 2 under on Windows NT version 3.5 the application runs correctly. If this application is run on Russian Windows version 3.1 with the Win32s libraries version 1.2 or 1.25 installed, some parts of the user interface appear as meaningless set of characters.

CAUSE

The strings in the resources are stored in UNICODE format. Yet you must pass all strings to Windows 3.1 in ANSI format. The drawing of the resources is done by Windows 3.1. Win32s simply reads the 32-bit resources, converts them to the 16-bit format equivalent, and passes the resources to Windows 3.1. One stage of the conversion is to convert the UNICODE strings into ANSI strings. This conversion for the Russian language is broken.

STATUS

Microsoft has confirmed this to be a problem in the product(s) listed at the beginning of this article. This bug will be fixed in the next release of Win32s.

Additional reference words: Cyrillic Russia 2.00 garbage corruption
KBCategory: kbbuglist
KBSubcategory: WIntlDev W32s

BUG: Can't Justify Text with Japanese Script Terminal Font

PSS ID Number: Q145755

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SYMPTOMS

If the Japanese Script of the Terminal font is selected into the device context and then SetTextJustification is called to justify a text string, the text is not justified.

CAUSE

The Japanese Script of the Terminal font has a tmBreakChar value of zero. This prevents SetTextJustification from specifying the amount of space Windows should add to the break characters in a string of text. The tmBreakChar value should be 32 which is the half-space character.

RESOLUTION

The Japanese Script of the Terminal font should not be used to create justified text using SetTextJustification. This problem will be fixed in the next release of Windows 95.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbprg kbbuglist
KBSubcategory: wintldev

BUG: ChooseColor Dialog Box Quits When Edit Control Subclassed

PSS ID Number: Q141202

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows NT version 3.51

SYMPTOMS

When the ChooseColor() API is called to display the custom color selection dialog box in an application that is globally subclassing edit controls, an application error (The memory could not be read) occurs in the application.

CAUSE

Certain messages are sent to this dialog before WM_INITDIALOG. These messages manipulate the dialog's edit controls. However, these messages assume the standard system edit control and act accordingly, thus causing the problem.

RESOLUTION

To avoid this error, applications that use a global customized edit control should use a superclassed edit control.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words:
KBCategory: kbprb kbbuglist
KBSubcategory:

BUG: ChooseFont() Sample Text Displays DBCS Chars Incorrectly

PSS ID Number: Q145756

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SYMPTOMS
========

The Font dialog's Sample text window displays DBCS characters incorrectly for the MSP-Gothic typeface font if the user selects a font style other than Regular.

CAUSE
=====

If the CHOOSEFONT structure's Flags field contains the CF_NOSCRIPTSEL flag when the ChooseFont() API is called and the user subsequently selects the MPS Gothic font in the Font list box and then sets the font style to a style other than regular, the DBCS text is displayed incorrectly. The CF_NOSCRIPTSEL flag is used in Windows 95 only and specifies that there should be no selection in the character set (Script) combo box.

Applications use this flag to support multiple character set selections. This flag is set on input only. When it is set, the script combo box is disabled and the lfCharSet member of the LOGFONT structure is set to DEFAULT_CHARSET.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

The occurance of this problem does not depend on any fonts or size of font. Therefore, the problem may occur while the sample text is displaying other fonts.

Additional reference words: 4.00
KBCategory: kbprg kbbuglist
KBSubcategory: wintldev

BUG: Console Applications Do Not Receive Signals on Windows 95

PSS ID Number: Q130717

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SYMPTOMS

Console applications call SetConsoleCtrlHandler() to install or remove application-defined callback functions to handle signals. On Windows 95, the signal handler function only gets called for the CTRL_C_EVENT and CTRL_BREAK_EVENT signals; the signal handler function is never called for the CTRL_SHUTDOWN_EVENT, CTRL_LOGOFF_EVENT, and CTRL_CLOSE_EVENT signals.

CAUSE

Windows 95 sends CTRL_C_EVENT and CTRL_BREAK_EVENT signals to console applications that have installed signal handers, but does not send CTRL_SHUTDOWN_EVENT, CTRL_LOGOFF_EVENT, or CTRL_CLOSE_EVENT signals.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: Win95 4.00 console event signal
KBCategory: kbprg kbbuglist
KBSubcategory: BseCon

BUG: Console Control Events May Be Missed

PSS ID Number: Q134284

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SYMPTOMS

In Windows 95, console applications that install console control handler functions by using SetConsoleCtrlHandler() do not always get all events when two or more events occur at almost the same time.

CAUSE

The Windows 95 console system does not queue up console control events. If multiple events occur rapidly in succession, events received later overwrite those received earlier, resulting in the earlier events being lost. The number of events that the console application receives depends on when the events actually arrive; the shorter the interval between the events, the more likely that one or more will be lost.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

The following code demonstrates the problem. It should receive five CTRL_C_EVENT and five CTRL_BREAK_EVENT events, and print a line for each. However, because events are not queued, it will print less than five of each event type, and may print only the last CTRL_BREAK_EVENT.

Code to Demonstrate Problem

```
// Console Application
#include <windows.h>
#include <stdio.h>

BOOL WINAPI CtrlHandler (DWORD dwEvent);

void main (void)
{
```

```
   printf ("Installing handler\n");
   SetConsoleCtrlHandler (CtrlHandler, TRUE);

   GenerateConsoleCtrlEvent (CTRL_C_EVENT, 0);
   GenerateConsoleCtrlEvent (CTRL_C_EVENT, 0);
   GenerateConsoleCtrlEvent (CTRL_C_EVENT, 0);
   GenerateConsoleCtrlEvent (CTRL_C_EVENT, 0);
   GenerateConsoleCtrlEvent (CTRL_C_EVENT, 0);

   GenerateConsoleCtrlEvent (CTRL_BREAK_EVENT, 0);
   GenerateConsoleCtrlEvent (CTRL_BREAK_EVENT, 0);
   GenerateConsoleCtrlEvent (CTRL_BREAK_EVENT, 0);
   GenerateConsoleCtrlEvent (CTRL_BREAK_EVENT, 0);
   GenerateConsoleCtrlEvent (CTRL_BREAK_EVENT, 0);

   printf ("Removing handler\n");
   SetConsoleCtrlHandler (CtrlHandler, FALSE);
}

BOOL WINAPI CtrlHandler (DWORD dwEvent)
{
   switch (dwEvent)
      {
      case CTRL_C_EVENT:
         printf("Got CTRL_C_EVENT\n");
         break;

      case CTRL_BREAK_EVENT:
         printf("Got CTRL_BREAK_EVENT\n");
         break;

      case CTRL_LOGOFF_EVENT:
         printf("Got CTRL_LOGOFF_EVENT\n");
         break;

      case CTRL_SHUTDOWN_EVENT:
         printf("Got CTRL_SHUTDOWN_EVENT\n");
         break;

      case CTRL_CLOSE_EVENT:
         printf("Got CTRL_CLOSE_EVENT\n");
         break;

      default:
         // unknown type--better pass it on.
         return (FALSE);
      }
   // Handled all known events
   return (TRUE);
}
```

Additional reference words: 4.00 Windows 95
KBCategory: kbui kbprg kbbuglist
KBSubcategory: BseCon

BUG: CopyFile Fails with Read-Only Files

PSS ID Number: Q145935

The information in this article applies to:

- Microsoft Win32 Software Development Kit for
  Microsoft Windows 95

SYMPTOMS

The CopyFile function fails and GetLastError returns ERROR_ACCESS_DENIED when you try to copy a read-only file to a core SMB server (such as a LAN Manager for UNIX server) or to a Netware server when the client computer uses real-mode network drivers. The destination file is created as a read-only file and is zero in length.

CAUSE

This error message occurs because there is a mismatch between the way that the Windows 95 redirector accesses files and the way that these servers expect this access.

RESOLUTION

Remove the read-only attribute before calling the CopyFile function to allow the file to be copied correctly. After the copy is complete, then the read-only attribute can be restored.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbprg kbbuglist
KBSubcategory: BseFileio

BUG: CreateDC Does Not Thunk DEVMODE Structure Correctly

PSS ID Number: Q128701

The information in this article applies to:

- Microsoft Win32s, versions 1.2 and 1.25

SYMPTOMS

Under Win32s version 1.2 and 1.25, when a Win32-based application displays a printer setup dialog box that calls PrintDlg with the PD_PRINTSETUP flag and the printing orientation is changed from Portrait to Landscape, the DEVMODE structure obtained from the PrintDlg call to CreateDC is passed and creates a printer DC. If the printer driver is a postscript driver, anything printed using this DC is still in Portrait mode.

CAUSE

The thunking layer for CreateDC does not thunk the DEVMODE structure for the postcript driver correctly.

RESOLUTION

To work around the problem, avoid calling CreateDC with a DEVMODE structure. Instead, directly create the pinter DC by calling PrintDlg with the PD_RETURNDC flag. Then change the printing orientation from within the print dialog box.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.20 1.25 3.10 3.50
KBCategory: kbprint kbbuglist
KBSubcategory: GdiPrn

BUG: CreateDC Fails in Windows NT-Based Service Application

PSS ID Number: Q137631

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SYMPTOMS
========

A CreateDC call fails in a Windows NT-based service application when printing to a network printer in Windows NT by clicking Printer and then Connect To on the Printer Manage menu.

WORKAROUND
==========

Create a local printer using the same printer model as the network printer and redirect the printer port to the network printer.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: service printing
KBCategory: kbprg kbprint kbbuglist
KBSubcategory: GdiPrn

BUG: CreateDirectoryEx() Returns Failure Although API Succeeds

PSS ID Number: Q140455

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

When CreateDirectoryEx uses the template directory parameter lpTemplateDirectory to create a directory, the function reports that it failed if the template directory string contains two back slashes (\\) at the end of the string.

For example, if the lpTemplateDirectory is set to C:\\, CreateDirectoryEx returns FALSE, and it sets the last error flag to (87L) ERROR_INVALID_PARAMETER, even though API successfully created the new directory.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words 4.00
KBCategory: kbprg kbbuglist
KBSubcategory:

BUG: CreateFile() Opens Read-Only File with Read/Write Access

PSS ID Number: Q140020

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SYMPTOMS
========

A Win32 application calls CreateFile() with GENERIC_READ and GENERIC_WRITE as the parameters for the desired access. The application tries to open an existing file on a Novell Netware server. This file has a read-only access permission set. The call to CreateFile() is successful.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

The information in this article is applicable only to the Microsoft Client for Netware for Windows 95. Ideally, in the previously described scenerio, the call to CreateFile() should fail with error 5L (ERRROR_ACCESS_DENIED), as it does on another Win32 platform - Windows NT. However, in Windows 95, an attempt to write to the opened file does fails. The condition described in this article applies only to files residing on Novell Netware Servers.

Additional reference words: 4.00
KBCategory: kbnetwork kbbuglist kb3rdparty
KBSubcategory: NtwkMisc BseFileio

BUG: DdeConnect Never Returns

PSS ID Number: Q136218

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS
========

A 32-bit application calls DdeConnect when all previous DDEML initialization has been performed. DdeConnect never returns.

CAUSE
=====

Any thread that creates a top-level window and doesn't have a message loop will cause DdeConnect to block. This is because DdeConnect calls SendMessage(HWND_BROADCAST...). In this call, SendMessage will put the message in the target thread's message queue and block the calling thread. If the target thread doesn't have a message loop, it will never process this message and therefore never return.

One complication is that some things create windows without the knowledge of the calling thread. It is known that some SQL, RPC, and DDEML function calls will do this. Any top-level window that is created by a thread that has no message loop will cause this to happen.

RESOLUTION
==========

The only current solution is to add a message loop to the thread that created the window.

STATUS
======

Microsoft has confirmed this to be a problem in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbui kbbuglist
KBSubcategory: UsrDde

BUG: Deadkey Not Accepted in Windows 95 Console Application

PSS ID Number: Q140456

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

Deadkeys cannot be accepted as input to a Windows 95 Console application even when the appropriate keyboard layout is selected. ReadConsoleInput() cannot read in deadkeys on Windows 95.

For example, pressing the circumflex (^) and then the letter e should produce an e with the circumflex on top. However, the combination keys do not print in a console application on a system running Windows 95 with a German keyboard selected.

CAUSE

Some of the code handling deadkey input to the console is missing from Windows 95.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbui kbbuglist
KBSubcategory:

BUG: DeletePrinter on Remote Printers Causes GPFs

PSS ID Number: Q151076

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  Microsoft Windows 95

SYMPTOMS

The DeletePrinter function in Windows 95 causes a General Protection Fault (GPF) if the printer is opened by specifying a UNC pathname.

Note that calling DeletePrinter on remote printers in Windows 95 is not supported. However, DeletePrinter should fail the call rather than cause a General Protection Fault.

RESOLUTION

Do not call DeletePrinter in Windows 95 on a printer handle that was opened by specifying a UNC path in the OpenPrinter function.

Applications should check the version of the operating system and prevent DeletePrinter from being called when the application is running on Windows 95 and the target printer is specified by a UNC path.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 Crash failure Spool32 GPF gp gp-f
KBCategory: kbprg kbbuglist
KBSubcategory: GdiPrn

BUG: ESC/ENTER Keys Don't Work When Editing Labels in TreeView

PSS ID Number: Q130691

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS

When editing labels in a TreeView control, you should be able to press the ESC key to cancel the changes or press the ENTER key to accept the changes. However, when the TreeView control is contained in a dialog box, IsDialogMessage processes the ESC and ENTER keystrokes and does not pass them on to the edit control created by the TreeView control, so the keystrokes have no effect.

CAUSE

The TreeView control creates and subclasses the edit control used for in-place editing. The subclass procedure does not process the WM_GETDLGCODE and WM_CHAR messages for the edit control properly.

RESOLUTION

To work around the problem, subclass the edit control and return DLGC_WANTALLKEYS in response to the WM_GETDLGCODE message. Then process the WM_CHAR messages for VK_ESCAPE and VK_RETURN.

To subclass the edit control, obtain the handle to the edit control by using the TVM_GETEDITCONTROL message, then use normal subclassing techniques. The control should be subclassed in response to the TVN_BEGINLABELEDIT notification. Remove the subclassing when the TVN_ENDLABELEDIT notification is received.

In response to the WM_CHAR|VK_ESCAPE message, have the application send the TVM_ENDEDITLABELNOW with fCancel = TRUE message to cancel the edit. In response to the WM_CHAR|VK_RETURN message, have the application send the TVM_ENDEDITLABELNOW with fCancel = FALSE message to accept the edit.

All other WM_CHAR messages should be passed on to the default edit control window procedure.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed

at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.30 4.00 95
KBCategory: kbui kbbuglist
KBSubcategory: UsrCtl W32s

BUG: EV_RING Isn't Detected in WaitCommEvent

PSS ID Number: Q137862

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

In a Win32-based environment, an application is supposed to detect a RING indication on a serial communications port by passing EV_RING as one of the flags in SetCommMask. After a RING has occurred, a call to WaitCommEvent should indicate that an EV_RING event has occurred. However, when this is done in Windows 95, WaitCommEvent never indicates that an EV_RING event occurred. In Windows NT, WaitCommEvent does indicate the EV_RING properly.

RESOLUTION
==========

There are two possible resolutions. One solution to this problem is to enable your modem device to indicate a ring by sending the string "RING" to the port. Your application can detect that a RING has occurred by reading the characters coming into the port and determining if the string "RING" has been received.

Another solution is to create a thread that repeatedly calls GetCommModemStatus to look for the MS_RING_ON flag to be set in the returned modem status. This method, however, severely degrades system performance.

The following code demonstrates this method:

```
   for (;;)
   {
      GetCommModemStatus(hCommPort, &dwModemStatus)
      if (MS_RING_ON & dwModemStatus)
         // RING has occurred
   }
```

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbbuglist
KBSubcategory: BseComm

BUG: FindFirstFile() Does Not Handle Wildcard (?) Correctly

PSS ID Number: Q130860

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

In Windows 95, the FindFirstFile() function interprets a wildcard (?) as "any character" instead of "zero or one character," its true meaning. This incorrect interpretation causes some searches to return invalid results. For example, if the files, TEMP.TXT and TEMPTEMP.TXT, are in the same directory, the following code finds the TEMPTEMP.TXT file, but not the TEMP.TXT file:

```
   HANDLE hFind;
   WIN32_FIND_DATA findData = {0};

   hFind = FindFirstFile ("TEM?????.???", &findData);

   if (hFind == INVALID_HANDLE_VALUE)
      MessageBox (hwnd, "FindFirstFile() failed.", NULL, MB_OK);
   else
   {
      do
      {
         MessageBox (hwnd, findData.cFileName, "File found", MB_OK);
      }
      while (FindNextFile(hFind, &findData));

      CloseHandle (hFind);
   }
```

Windows NT correctly finds both the TEMP.TXT and TEMPTEMP.TXT files.

RESOLUTION
==========

To work around this problem, choose an alternative wildcard search and apply further processing to eliminate files that are found by the alternative search, but do not match the original search. For example, the code above could be changed to search for TEM*.* instead of TEM?????.???. Then you could make an additional test for filenames that are up to 8 characters in length, followed by a ".", followed by up to 3 more characters (8.3).

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 95 4.00 regular expression wild
KBCategory: kbprg kbbuglist
KBSubcategory: BseFileio

BUG: GetCommProperties Leaks a Handle

PSS ID Number: Q139657

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SYMPTOMS

Repeated calls to GetCommProperties cause the process handle count to increase and never decrease. This can be detected by running PerfMon. This eventually causes system failures.

RESOLUTION

To work around this problem, don't call GetCommProperties repeatedly. Call GetCommProperties once, and then store the result for use later. If you must repeatedly call GetCommProperties, you should periodically exit the process and restart. You may want to create a separate, short-lived process to get the desired information and communicate it to the main process by using IPC methods.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words:
KBCategory: kbprg kbbuglist
KBSubcategory: BseCommapi

BUG: GetDiskFreeSpace() Fails with a UNC Path to Network Share

PSS ID Number: Q137230

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 4.0

SYMPTOMS
========

The GetDiskFreeSpace() function fails with a universal naming convention (UNC) path to a network share. This failure causes problems for various applications. For example, WinHelp can neither load nor create a full-text index through a UNC path. The same operation works fine using a drive-relative path.

This problem does not occur under Microsoft Windows NT. However, you must include a trailing backslash, so that the UNC path looks like the following:

```
\\server\share\
```

RESOLUTION
==========

Use a drive letter instead. For example:

```
GetDiskFreeSpace(<UNC name>)
if <GetDiskFreeSpace fails>
   WNetAddConnection()    // Connect and get a drive letter.
   GetDiskFreeSpace(<drive letter>:)
   WNetCancelConnection() // Disconnect.
```

NOTE: A trailing backslash is required for the drive letter under Microsoft Windows 95. Under Microsoft Windows NT, the call succeeds with or without the trailing backslash.

STATUS
======

Microsoft has confirmed this to be a bug in Microsoft Windows 95. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 95 buglist4.00
KBCategory: kbprg kbbuglist
KBSubcategory: BseFileio

BUG: GetKerningPairs Sometimes Fails on Win32s 1.2 and 1.25a

PSS ID Number: Q125872

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Win32s versions 1.2 and 1.25a

SYMPTOMS

GetKerningPairs will sometimes fail on Win32s version 1.2 causing the 32-bit application to exit mysteriously. The problem may only occur once in a while with many successful runs interrupted by a single unsuccessful run.

CAUSE

The thunking layer for GetKerningPairs contains a bug in the code that allocates a temporary buffer passed to the 16-bit version of GetKerningPairs. The errant code executes whenever the number of kerning pairs requested is equal to or less than 128. Requesting GetKerningPairs to return 129 or more kerning pairs forces the thunking layer to use an alternative buffer allocation method.

RESOLUTION

To work around the problem, ensure that the number of kerning pairs requested from GetKerningPairs is greater than 128.

Typically, kerning pairs are retrieved with two calls to GetKerningPairs. The first call retrieves the number of kerning pairs available. A buffer is allocated based on the number of pairs returned. Then the second call to GetKerningPairs retrieves the kerning pairs into the buffer.

To avoid the bug in GetKerningPairs, follow these steps:

1. Retrieve the number of kerning pairs available from GetKerningPairs.

2. Check that this value is greater than 128. If it is less than or equal to 128, reset the variable to an arbitrary value greater than 128 -- like 129.

3. Use the new value to allocate the buffer of KERNINGPAIRS and pass this new value with the buffer to the second GetKerningPairs call.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will

post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.20 font kerning
KBCategory: kbgraphic kbbuglist
KBSubcategory: W32s GdiFnt

BUG: GetObject Returns Partial Information for DIBSections

PSS ID Number: Q151072

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  Microsoft Windows 95

SYMPTOMS

The dshSection and dsOffset members of the DIBSECTION structure contain NULL when the GetObject function is called on a file-mapped DIBSection. These structure members should contain the handle to the file-mapping object and the offset that were passed into the original CreatDIBSection function call. Note that this is a problem only when DIBSections are created from file-mapping objects. Normally, NULLs are used in the hSection and dwOffset parameters of the CreateDIBSection function to ask Windows to allocate the DIBSection's memory. Under these conditions, GetObject would be expected to return NULL for these structure members.

RESOLUTION

To work around this problem, applications must cache the values passed to the CreateDIBSection function.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 DIB CreateFileMapping MapViewOfFile
KBCategory: kbprg kbgraphic kbbuglist
KBSubcategory: GdiBmp

BUG: GetService Gives Only Local Service Names

PSS ID Number: Q138039

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0

SYMPTOMS

An application's attempt to use name registration and the resolution API GetService is unsuccessful. Service names, other than those on the local computer, cannot be found.

CAUSE

GetService does not work for both Windows NT build 1057 and Windows 95 build 950.6. GetService is designed to enumerate through all name space providers to query a service. However, none of the providers currently implement GetService helper functions.

WORKAROUND

Use GetAddressByName. Alternately, for NetWare services, send SAP packets manually using WinSock. Use gethostbyname for TCP/IP services.

The best workaround is to call GetAddressByName, which can obtain addresses for TCP/IP services, NetWare services and NetBIOS names. GetAddressByName takes service names and global unique identifiers (GUID) just as GetService does, and it returns the associated address for that service.

Another workaround is to query each name space independently. For example, you might send out a SAP query to NetWare services for services registered in the Bindery or NDS for a NetWare server. To learn more about SAP queries, refer to the IPX router specification provided by Novell.

To find the IP address of a TCP/IP service, you can use gethostbyname in the WinSock API. Your service must be registered in the DNS database.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbnetwork kbprg kbbuglist
KBSubcategory:

BUG: getsockopt for IPX_MAX_ADAPTER_NUM Fails on Windows 95

PSS ID Number: Q139131

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

When an application calls getsockopt with option IPX_MAX_ADAPTER_NUM on Windows 95, the call fails with Winsock error 10042 (Bad Protocol Option).

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

The following article in the Microsoft Knowledge Base illustrates a way to use getsockopt() and IPX_MAX_ADAPTER_NUM to enumerate addresses for Winsock families other than IP. This example works in Windows NT, but because of the problem documented in this article, it will not work in Windows 95

   ARTICLE-ID: Q129315
   TITLE   : How to Use WinSock to Enumerate Addresses

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkMisc

BUG: GetTextExtentPoint Fails in App Based on Windows/Win32s

PSS ID Number: Q147647

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Application Programming Interface included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.30c

SYMPTOMS

GetTextExtentPoint fails when all of the following conditions are present:

- You have either a 16-bit application running under Windows 3.1 or a 32-bit application running under Win32s.
- The application uses GetTextExtentPoint().
- The string buffer parameter has been allocated with GlobalAlloc() or has been reallocated with GlobalReAlloc().
- The string size is an exact multiple of 32 bytes.

Under Win32s, GetLastError returns an error 87 (Invalid Parameter). The debug version of Windows 3.1 displays a Fatal Exit code 0x700A with this error message:

```
GetTextExtentPoint - Invalid string
```

CAUSE

As stated earlier, this problem occurs only when the allocated size of the string is an exact multiple of 32 bytes.

This is a bug in Microsoft Windows 3.1 that causes ramifications in a 16-bit application running under Windows 3.1. A 16-bit application running under Windows NT 3.51 will have the same problem, but it will run without any problems under Windows 95.

This bug is not specific to Win32s. However, because Win32s runs under the Windows environment, a 32-bit application running under Windows 3.1 by way of Win32s will also exhibit this problem. However, a 32-bit application running under Windows NT or Windows 95 will not exhibit this problem.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed

at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

Note that GetTextExtentPoint() will succeed if you pass a string constant that has a size that is a multiple of 32. For example, the following will work:

```
   GetTextExtentPoint(hDC, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 32, &size);
```

Sample Code to Reproduce Problem

Note that error checking is not implemented in the following code.

```
 HGLOBAL hMem;
 HDC hDC;
 SIZE size;
 int blksize, i;
 char szMsg[256];
 LPSTR lpBuf;

  blksize=32;
  #ifdef WIN32
    SetLastError(0);
  #endif

  hDC = GetDC(hWnd);
  hMem = GlobalAlloc(GHND, blksize);
  lpBuf = (LPSTR) GlobalLock(hMem);
  for (i=0; i<blksize; i++)
    lpBuf[i] = 't';

  if (!GetTextExtentPoint(hDC, lpBuf, blksize, &size))
  {
     #ifdef WIN32
       wsprintf(szMsg, "GetTextExtentPoint Error %d\r\n", GetLastError());
     #else
       wsprintf(szMsg, "GetTextExtentPoint Error\r\n);
     #endif
     MessageBox(NULL, szMsg, "Test", MB_OK);
  }
  GlobalUnlock(hMem);

  #ifdef WIN32
    SetLastError(0);
  #endif

  hMem = GlobalReAlloc(hMem, 2*blksize, GMEM_ZEROINIT);
  lpBuf = (LPSTR) GlobalLock(hMem);
  for (i=0; i<blksize; i++)
    lpBuf[blksize+i] = 'T';
```

```
  if (!GetTextExtentPoint(hDC, lpBuf, 2*blksize, &size))
  {
     #ifdef WIN32
       wsprintf(szMsg, "GetTextExtentPoint Error %d\r\n", GetLastError());
     #else
       wsprintf(szMsg, "GetTextExtentPoint Error \r\n");
     #endif
     MessageBox(NULL, szMsg, "Test", MB_OK);
  }

  GlobalUnlock(hMem);
  if (hMem) GlobalFree(hMem);
```

Additional reference words: win32s gdi 1.30c 4.00 3.10 3.51
KBCategory: kbprg kbbuglist
KBSubcategory: w32s kbgdi

BUG: ImmSetConversionStatus Doesn't Work When IME Is Closed

PSS ID Number: Q150024

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Software Development Kit (SDK) for:

    - Microsoft Windows NT, version 3.51
    - Microsoft Windows 95, version 4.0

SYMPTOMS
========

Under Microsoft Japanese Windows 95 and Microsoft Japanese Windows NT 3.51, ImmSetConversionStatus doesn't work if the native IME -- MS IME 95 -- is used when IME is closed.

If the IME is closed and ImmSetConversionStatus is called to set the conversion status, the new conversion status is not used when the IME is opened again. Instead, the conversion status reverts to the old setting, although the IME status window shows the new setting.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

Steps to Reproduce Problem
---

1. Open MS IME 95. Set it to double-byte Hiragana mode, and type in some Hiragana characters. Then close the IME.

2. Call ImmSetConversionStatus to set the IME to Katakana mode.

3. Open IME. The status window appears to be in Katakana mode.

4. Start typing, and notice that the Status Window reverts back to double-byte Hiragana mode.

Additional reference words: 4.00 3.51 DBCS IMM IME
KBCategory: kbprg kbbuglist
KBSubcategory: wintldeb

BUG: Incomplete IPX_ADDRESS_DATA Structure Under Windows 95

PSS ID Number: Q139130

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

When an application calls getsockopt with option IPX_ADDRESS on Windows 95, the IPX_ADDRESS_DATA structure is only partially filled. The only parts of the IPX_ADDRESS_DATA structure that are filled in are the UCHAR netnum[4] and UCHAR nodenum[6] fields.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkMisc

BUG: Incorrect Painting Using Pattern Brushes on DIBSections

PSS ID Number: Q149956

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 4.0

SYMPTOMS

When using a pattern brush from CreatePatternBrush() on a DIBSection, where the DIBSection is of a different bit-depth than the screen, the result of the painting is incorrect.

RESOLUTION

A pattern brush, created with a call to CreatePatternBrush(), can only be used on device dependent bitmaps (DDBs) or DIBSections of the same bit-depth as the screen. A DIBPatternBrush should be used on a DIBSection having a different bit-depth than the screen. This DIBPatternBrush is created with a call to CreateDIBPatternBrushPt().

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 DIB brush pattern
KBCategory: kbgraphic kbbuglist
KBSubcategory:

BUG: LoadString Returns Wrong Number of Chars for DBCS Strings

PSS ID Number: Q140452

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95
- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

Under the Japanese version of Windows 95, the LoadString API returns an incorrect number of single-byte characters when loading a string that meets both of these conditions:

- The string contains at least one DBCS character.
- The string resource contains more than cchBuffer bytes.

CAUSE

The LoadString API returns the number of single-byte characters into the buffer. However, when it loads a DBCS string that is longer than the given maximum size, LoadString returns an incorrect value. LoadString loads cchBuffer characters and returns the length in bytes. LoadString should load cchBuffer bytes, not characters. The return value will be equal to or greater than cchBuffer. The resulting string copied into the buffer is not properly null terminated.

In Windows NT (Unicode), the API returns the number of characters. In Windows 95 (ANSI version), the API returns the number of bytes. Likewise, the cchBuffer parameter is characters for Windows NT and bytes for Windows 95.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbprg kbbuglist
KBSubcategory:

BUG: MCI Open of MIDI File with Installable I/O Proc Fails

PSS ID Number: Q142894

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK) for:
  - Windows NT versions 3.5 and 3.51
  - Windows 95

SYMPTOMS

On various Windows platforms, using MCI to open a MIDI file element with the FOURCC of an installed I/O procedure as part of the element name fails according to the following table:

Platform	EXE	MCI Open Results
Windows 3.1	16-bit	fails, error #1
Win32s	32-bit	fails, error #1
Windows 95	16-bit	ok
Windows 95	32-bit	fails, error #2
Windows NT 3.51	16-bit	fails, error #1
Windows NT 3.51	32-bit	fails, error #1

error #1 "MCI Error: Cannot find the specified file. Make sure the path and filename are correct."

error #2 "MMTask caused an invalid page fault in module <unknown> at 0000:1000100A"

CAUSE

MCI uses the Sequencer device to open and play MIDI file elements. Mciseq.drv is the system component that contains the MCI MIDI sequencer functionality. In Windows 3.1 and Windows NT 3.51, Mciseq.drv has a bug that causes the user-supplied FOURCC to be overriden. Mciseq.drv always sets the fccIOProc field of the MMIOINFO structure that it passes to mmioOpen() to FOURCC_DOS instead of the user-supplied FOURCC. This causes mmioOpen() to use the built-in MS-DOS file system I/O procedure instead of the user-installed I/O procedure. The result is the error "MCI Error: Cannot find the specified file. Make sure the path and filename are correct."

Although this bug is fixed for 16-bit applications in Windows 95, a limitation in the Windows 95 system architecture affects 32-bit application code that attempts this operation. In the 32-bit case, attempting to play a MIDI file using MCISEQ.DRV and an installed I/O procedure results in the

error "MMTask caused an invalid page fault in module <unknown> at 0000:1000100A".

RESOLUTION

There is no workaround in Windows 3.1 or Windows NT 3.51 that will allow MCI to open and play a MIDI file element using an installed I/O procedure. Therefore alternative solutions should be pursued on these platforms. For example, in the case of playing a MIDI application resource using MCI, you can work around the installed I/O procedure problem by first copying the MIDI resource to a temporary file and then using MCI to play the file element from disk with the default I/O procedure.

Because 16-bit application code in Windows 95 is not affected by the system architecture limitation that does affect 32-bit code in Windows 95, a 32-bit application based on Windows 95 can thunk to a 16-bit DLL to play back a MIDI file using MCI and an installed I/O procedure.

STATUS

Microsoft has confirmed these problems as bugs in the Microsoft products listed at the beginning of this article. We are researching these problems and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.10 3.50 4.00 mmio ioproc
KBCategory: kbmm kbbuglist
KBSubcategory: MMMidi

BUG: MCI Unable to Open .AVI File Using File-Interface Pointer

PSS ID Number: Q140750

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

When you attempt to use MCI to access an .avi file by using its reference pointer returned from AVIFileOpen(), the following error message occurs under Windows 95:

   mmsystem 275: CANNOT FIND THE SPECIFIED FILE, MAKE SURE THE FILE AND PATHNAME ARE CORRECT"

This problem does not occur under Windows NT or Windows 3.11.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

Code to Demonstrate the Problem

```
hrReturnVal = AVIFileOpen(&pfile,
                          "c:\\winvideo\\sample.avi",
                          OF_SHARE_DENY_WRITE,
                          0L);

if (hrReturnVal == 0)
{
     wsprintf(ach, "AVIVideo!@%ld", pfile);
     MCIWndOpen(hwndMci, ach, 0);      //Bug occurs here.
     MCIWndPlay(hwndMci);
}
```

Additional reference words: 4.00 file-interface stream-interface pointer
KBCategory: kbmm kbprg kbbuglist
KBSubcategory: MMVideo

BUG: MCIWndGetPositionString Makes Time Format Milliseconds

PSS ID Number: Q135304

The information in this article applies to:

- Microsoft Video for Windows Development Kit version 1.1
- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, 4.0

SYMPTOMS

Calling MCIWndGetPositionString causes the time format (set with MCIWndSetTimeFormat) to change to milliseconds. This applies only to devices that have tracks.

WORKAROUND

To work around the problem, reset the time format after calling MCIWndGetPositionString as follows:

```
#ifdef _WIN32

TCHAR pszFormat[30];  // Use a TCHAR in Win32
TCHAR pszPosition[30];

#else  // Not Win32

char pszFormat[30];
char pszPosition[30];
#endif

   // Get the time format - hWnd is the hWnd of the MCIWnd
   MCIWndGetTimeFormat(hWnd, pszFormat, sizeof(pszFormat));

   // Get the position - hWnd is the hWnd of the MCIWnd
   MCIWndGetPositionString(hWnd, pszPosition, sizeof(pszPosition));

   // Reset the time format - hWnd is the hWnd of the MCIWnd
   if (0 == MCIWndSetTimeFormat(hWnd, pszFormat))
      // Successful
   else
      // Could not reset time format
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.50 3.10 4.00
KBCategory: kbmm kbbuglist kbprg kbcode
KBSubcategory: MMVideo MMMisc MCIWnd

BUG: MIDL compiler structure packing problems

PSS ID Number: Q136500

The information in this article applies to:

- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

An RPC application that uses stub code generated by MIDL throws memory exceptions, causes a general protection (GP) fault, or overwrites memory in the data heap. The RPC application is using a structure or union in the IDL file.

CAUSE

The /Zp (packing) option of the MIDL compiler does not affect the size of the structure; MIDL always uses a packing of 1. However, the /Zp option of the C compiler does increase the size of the structure. Because MIDL assumes a packing of one, it generates code that allocates less memory than what is actually needed.

You can verify the behavior of the MIDL compiler by searching your client stub code for _StubMsg.pfnAllocate. This call is present only with semi-interpreted stubs (the default MIDL option, /Os). Try compiling your IDL file with different /Zp settings, and notice that the memory allocated by _StubMsg.pfnAllocate is unchanged, though it should change to accommodate packing space.

RESOLUTION

There are three solutions to correct this problem:

1. Use MIDL's /Oi option.

   -or-

2. Use #pragma pack(1) in the stub code to use a packing of 1.

   -or-

3. Pad your interface declaration to match the padding done by the C compiler.

Please see MORE INFORMATION for details.

STATUS

Microsoft has confirmed this to be a problem in MIDL 2.00.0102 for the

Win32 SDK. This problem has been fixed in MIDL 3.0, now shipping with Windows NT 4.0 beta SDK.

MORE INFORMATION
================

Fully interpreted stubs (MIDL option /Oi) use another approach for parameter marshaling, and this alternative approach does not have the problem with the packing. The stub code produced with /Oi is slightly slower, but for most applications the loss is negligible. If you decide to use the /Oi option, make sure your remote procedures are declared as __stdcall. Refer to the MIDL documentation for more information on /Oi.

If you are not interested in using the /Oi option, you can manually pad your interface declaration. To manually pad your interface declaration, you must first determine the packing used by the C compiler. Visual C++ 2.x for the x86 uses a packing of 4 by default. Otherwise, the packing is specified by /Zp or #pramga pack(). Each element must start on a multiple of the packing size.

The following example shows how to pad a structure with a packing of 4.

```
typedef struct   // This struct goes in the C code
{
   char c;
   long l;
} PADDED_BY_COMPILER_STRUCT;

typedef struct   // This struct goes in the IDL file
{
   char c;
   char pad[3];  // Add 3 bytes to align the next field on byte 4
   long l;
} MANUALLY_PADDED_IDL_STRUCT;
```

Another alternative is to wrap the stub code generated by MIDL with

```
#pragma pack(1)
```

Each time MIDL generates the stub code, you must insert #pragma pack statements into the server stub.

For example:

```
/* File created by MIDL compiler version 2.00.0102 */
/* at Fri Aug 18 13:48:23 1995
 */
//@@MIDL_FILE_HEADING(  )

#pragma pack(1)         // Added to fix MIDL pack problem

#include <string.h>
#include "bug.h"

... code generated by MIDL ...
```

```
   #pragma pack()          // Added to revert to default packing
-------------------------------------------------------------------------

Additional reference words: 3.50 4.00 95 2.00.0102
KBCategory: kbnetwork kbbuglist kbtool
KBSubcategory: NtwkRpc
```

BUG: Multiple Definitions for EM_CHARFROMPOS & EM_POSFROMCHAR

PSS ID Number: Q137249

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS
========

If you are using the EM_CHARFROMPOS and EM_POSFROMCHAR messages with a RichEdit control as outlined in the Win32 SDK documentation, and they are not working, it is likely that your application is using incorrect values for EM_CHARFROMPOS and EM_POSFROMCHAR.

CAUSE
=====

The definitions for EM_CHARFROMPOS and EM_POSFROMCHAR in Richedit.h are different from the definitions in Winuser.h.

RESOLUTION
==========

The programmer needs to use the definitions from Winuser.h as these work correctly. Include Winuser.h before Richedit.h. If these messages still don't work, the message values should be hard-coded to those from Winuser.h.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.30 4.00
KBCategory: kbui kbbuglist
KBSubcategory: UsrCtl

BUG: NetBIOS Find Name Does Not Work for IPX/SPX and NetBEUI

PSS ID Number: Q137916

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

If an application does a NetBIOS find name (NCBFINDNAME) on a LANA that corresponds to the NetBEUI protocol, the command fails and the error code returned is 0x03 (Invalid command).

If the LANA number corresponds to the IPX/SPX protocol, then although a valid FIND_NAME_BUFFER is returned, all the fields are set to zero.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

Using the LANA number that corresponds to the TCP/IP protocol returns expected results.

REFERENCES

IBM Local Area Network Technical Reference #SC30-3382-2.

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbtshoot kbbuglist
KBSubcategory: NtwkNetbios

BUG: NetBIOS Returns Wrong Adapter Type for NetBEUI

PSS ID Number: Q137915

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

When an application calls Adapter Status (NCBASTAT) on a LANA number that corresponds to the NetBEUI protocol, the adapter_type field of the ADAPTER_STATUS structure for an ethernet card is incorrectly set to 0xFF.

This is interpreted to be a Token Ring card.

If a LANA number is used that corresponds to any other protocol (for example, TCP/IP or IPX/SPX), the adapter_type is correctly reported as Ethernet.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

REFERENCES

IBM Local Area Network Technical Reference #SC30-3382-2.

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbtshoot kbbuglist
KBSubcategory: NtwkNetbios

BUG: Pressing SHIFT+ESC Doesn't Generate WM_CHAR on Windows 95

PSS ID Number: Q129861

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SYMPTOMS

Pressing SHIFT+ESC in an application running under Windows 95 doesn't generate a WM_CHAR message even though it does generates a WM_CHAR message for applications running under Windows version 3.x and Windows NT.

CAUSE

The key table that TranslateMessage on Windows 95 uses to generate the WM_CHAR messages doesn't include the SHIFT+ESC key combination.

RESOLUTION

If you need to use this key combination, use the WM_KEYDOWN or WM_KEYUP messages.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available. This bug is scheduled to be corrected in a future version of Windows 95.

Additional reference words: 4.00
KBCategory: kbui kbbuglist
KBSubcategory: UsrInp

BUG: Printer Port Redirection Not Effective for Windows Apps

PSS ID Number: Q152788

The information in this article applies to:

 - Microsoft Win32 Application Programming Interface included with:
   - Microsoft Windows 95

SYMPTOMS
========

Changing the redirection of an MS-DOS LPT device that is used by Windows fails. This problem occurs when using the WNetCancelConnection and WNetAddConnection functions to change a previously-redirected LPT port. Although the functions succeed and Windows reports the new device redirection, any Windows printer driver using the device continues to print to the original remote printer.

Similar behavior occurs if the Capture Printer Port feature is used in the printer properties dialog. Any attempt to change a printer port's connection to a different remote printer will appear to succeed but the print jobs arrive at the original remote printer.

This problem occurs only with Windows applications. MS-DOS applications executed within a console or "DOS" box are not affected by this problem and their output will be properly directed to the network print share for the LPTX device.

RESOLUTION
==========

To work around this problem, do not use WNetAddConnection and MS-DOS devices to connect Windows printer drivers to network print shares. Instead, configure Windows printers to use print shares directly via network ports. This behavior is accomplished by using the Print Spooler functions: OpenPrinter, GetPrinter, SetPrinter, ClosePrinter, EnumPorts, and AddPort. Continue to use the WNet functions in Windows to redirect MS-DOS devices for MS-DOS applications.

The Print Spooler is a new feature of Windows 95 and Windows NT. For more information on using the Print Spooler APIs to configure network printers, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q152551
   TITLE   : How to Connect Local Printers to Network Print Shares

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 network print queue spooler
KBCategory: kbprg kbprint kbbuglist
KBSubcategory: NtwkMisc

BUG: Problem Setting a Systemwide WH_CALLWNDPROC Hook

PSS ID Number: Q149862

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 4.0

SYMPTOMS

Under both Windows NT and Windows 95, a Win32 application can set a system wide WH_CALLWNDPROC hook using the SetWindowsHookEx() API. This hook allows the application to examine messages being sent by any process in the system using the SendMessage() API. The DLL containing the hook callback procedure is automatically injected into the address spaces of any process that calls SendMessage(). Under WINDOWS 95, pressing the ALT+ESC keys when the hook is installed injects the DLL into the KERNEL32.DLL process.

When the application later uninstalls the hook by calling UnHookWIndowsHookEx(), the DLL is unmapped from the address spaces of all the processes, except KERNEL32.DLL, in the system where it was injected. This leads to problems when you try to copy over or delete the DLL from the machine. These operations cause an error because the DLL file is considered to be in use by KERNEL32.DLL.

RESOLUTION

The only way to resolve this problem currently is to restart the machine.

STATUS

Microsoft has confirmed this to be a bug in the products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbprg kbbuglist
KBSubcategory:

BUG: QueryPerformanceFrequency Returns Wrong Value on PC 9800

PSS ID Number: Q152145

The information in this article applies to:

- Microsoft Win32 Software Development Kit for
  - Microsoft Windows 95, version 4.0
  - Microsoft Windows NT, version 3.51

SYMPTOMS

QueryPerformanceFrequency() called from an application running on an NEC PC9800 Series computer with Japanese Win95 (PC9800 version), returns a static value (1193180 Ticks/sec) regardless of the actual frequency of the high-resolution performance counter. On NEC PC 9800 Series computers, the static value can be 2457600 ticks/sec.

RESOLUTION

The same API returns the correct value on NEC PC9800 Series computers running Windows NT PC 9800 version.

This problem does not exist on any other platforms.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.51 4.00
KBCategory: kbprg kbbuglist
KBSubcategory: wintldev

BUG: Race Between 2 Threads Sharing a Socket Causes Problem

PSS ID Number: Q126346

The information in this article applies to:

- Microsoft Win32 Device Development Kit (DDK) for Windows NT version 3.5

SYMPTOMS

Packets between the TCP, UDP, and IP layers are lost.

CAUSE

There appears to be a problem with a race condition between two threads that share a socket where one is closing a socket while the other tries to call recvfrom() on the same socket. This causes problems the next time a socket is bound to the same UDP port.

RESOLUTION

The vendor should implement a workaround within the application so that this race condition does not occur.

STATUS

Microsoft has confirmed this to be a problem in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

This Sockets/UDP problem was discovered while testing the TX1000 NCPI driver for Windows NT. Here are some notes showing what appears to be happening:

```
Thread 1                        Thread 2
--------                        ---------
Create DGRAM socket x
Bind socket x to:
        IPADDR = ANY
        PORT = 1571
Create thread 2 ----------------->
                                RecvFrom on socket x
                                ...
                                Packet received on x (recvfrom completes)
                <---------------signal main thread
```

```
Process rec'd packet            wait for main thread to consume buffer
Signal buffer available--------->
                                RecvFrom on socket x
                                ...  { repeats many times }
```

Normal Shutdown Sequence (on Last Packet)

```
                                Packet received on x (recvfrom completes)
                <---------------signal main thread
Process rec'd packet            wait for main thread to consume buffer
Signal buffer available--------->
Application done                (1) RecvFrom on socket x
(2) close socket x
                                (3) RecvFrom fails with expected error
                                Thread terminates
```

Usually, events occur in sequence (1, 2, then 3). In this normal case, the socket is cleared correctly, and everything works the next time the application runs.

Shutdown Sequence that Causes Problems (on Last Packet)

```
                                Packet received on x (recvfrom completes)
                <---------------signal main thread
Process rec'd packet            wait for main thread to consume buffer
Signal buffer available--------->
Application done
(1) close socket x              (2) RecvFrom on socket x
                                (3) RecvFrom fails with expected error
                                Thread terminates
```

In this case, the sequence is slightly different. The closesocket() function from main thread starts, but does not complete, before thread 2 runs. While thread 1 is suspended awaiting completion of closesocket(), thread 2 runs and posts next recvfrom() on same socket. The closesocket() function completes successfully, and recvfrom() fails as in normal case. But the next time the application runs and binds to the same UDP port, the following occurs:

- All socket calls (bind(), recvfrom()) are successful

- All incoming packets to that UDP port are discarded before reaching the application -- recvfrom() never completes. The following command shows that the "receive errors" count has been incremented once for each incoming packet that was lost:

  ```
  netstat -s -p udp
  ```

The conclusion is that in this case the socket was not cleaned up properly due to the race condition between the closesocket() and the recvfrom() functions.

Additional reference words: 3.50

KBCategory: kbnetwork
KBSubcategory: NtwkWinsock

BUG: ReadFile Returns Wrong Error Code for Mailslots

PSS ID Number: Q139715

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

If a server opens a Mailslot using CreateMailslot(), specifies a timeout, and then uses ReadFile to recieve data, the ReadFile will fail if there is no data available. GetLastError() returns an error code of 5 (access denied).

RESOLUTION
==========

Interpret this error code to mean that no data is available.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbbuglist
KBSubcategory:

BUG: Reading Past End of Volume with FAT File Produces Errors

PSS ID Number: Q150860

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  - Microsoft Windows NT version 3.51

SYMPTOMS
========

Windows NT allows access to partitions by opening a volume using the "\\.\x:" nomenclature. This procedure allows a process to access a partition directly. Reading beyond the end of a volume formatted with the New Technology File System (NTFS) causes the read operation to truncate. Subsequent read operations succeed, but the amount read is set to zero bytes, which indicates the End-Of-File (EOF) condition. Reading beyond the end of a volume formatted with the FAT file system causes the read operation to fail and GetLastError to return ERROR_INVALID_PARAMETER.

RESOLUTION
==========

To work around this behavior, do not perform read operations that extend beyond the length of the volume. The length of the volume is determined by examining the results of the DeviceIoControl function when called with the IOCTL_DISK_GET_PARTITION_INFO control code. The PartitionLength member of the PARTITION_INFORMATION structure filled out by the IOCTL contains a pair of DWORDS that indicate the size of the volume in bytes.

STATUS
======

Microsoft has confirmed this to be a problem in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.51
KBCategory: kbprg kbbuglist
KBSubcategory: BseFileio

BUG: Rich Edit Control Improperly Displays Print Preview

PSS ID Number: Q142320

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS
========

When you implement print preview using a rich edit control, the preview looks incorrect. For example, the text is the wrong size or is not placed at the correct location. A common manifestation of this bug is to have the text extend beyond the end of the printable area.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.30 4.00 1.0 RTF RichEdit WordPad
EM_FORMATRANGE
KBCategory: kbgraphic kbbuglist
KBSubcategory: UsrCtl

BUG: RTS_CONTROL_TOGGLE Doesn't Cause RTS to Toggle

PSS ID Number: Q140030

The information in this article applies to:

- Microsoft Win32 Software Development Kit for Windows 95

SYMPTOMS
========

Setting the fRtsControl member of the DCB structure to RTS_CONTROL_TOGGLE should cause the RTS line to go high when data is available for sending. When no more bytes are available for transfer, the RTS line should be set to low. In Windows 95, the RTS line stays high.

RESOLUTION
==========

To work around this problem, use EscapeCommFunction to change the state of the RTS line manually. Because the system doesn't do the RTS manipulation, the application should implement the functionality itself. When data is about to be sent, the application should set RTS high. When all of the data has been sent, RTS should be set to low. This is only necessary when the device being communicated with requires it. For more information, see the Win32 documentation for help on EscapeCommFunction.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbprg kbbuglist
KBSubcategory: BseCommapi

BUG: Screen Saver Not Getting WM_DESTROY in Windows NT

PSS ID Number: Q140727

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows NT versions 3.5, 3.51

SYMPTOMS

When a 32-bit screen saver running under Windows NT 3.5 or 3.51 terminates due to a mouse move, mouse click, or key press, although the application terminates, the window never gets the WM_DESTROY message. Failure to receive this message poses a problem as applications are not able to call clean-up routines such as deleting GDI objects, freeing memory objects, and so on. This results in a memory leak and further reduction of system resources each time the screen saver application is run.

This problem does not occur when the screen saver is run in Test mode under Windows NT's Control Panel. In Test mode, the screen saver receives the WM_DESTROY message.

STATUS

Microsoft has confirmed this to be a problem with the Scrnsave.lib file that shipped with the Win32 SDK for Windows NT 3.51 and Windows 95. It is scheduled to be fixed in the next release of the Win32 SDK. However, this problem is specific to Windows NT; it does not occur with 32-bit screen savers under Windows 95.

Additional reference words: 3.50
KBCategory: kbgraphic kbbuglist
KBSubcategory: GdiScrSav

BUG: SetStretchBltMode() Ignored

PSS ID Number: Q138105

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

On Windows 95, calling SetStretchBltMode() does not change the behavior of StretchBlt(). StretchBlt() continues to operate as if the stretching mode were STRETCH_DELETESCANS or COLORONCOLOR.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 GetStretchBltMode SetStretchBltMode StretchDIBits STRETCH_ANDSCANS BLACKONWHITE STRETCH_HALFTONE HALFTONE STRETCH_ORSCANS WHITEONBLACK
KBCategory: kbgraphic kbbuglist
KBSubcategory:

BUG: SetWindowPlacement and ptMin.x or ptMax.x = -1

PSS ID Number: Q110793

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

Passing an X coordinate value of -1 to SetWindowPlacement causes the parameter to be ignored. If ptMinPosition.x is set to -1, SetWindowPlacement won't reset the minimized window coordinate; this is also true for ptMaxPosition.x. This was undocumented in the Windows 3.1 SDK documentation.

CAUSE

This problem is caused by the use of -1 as a special value. A value of -1 in the X coordinate causes the API (application programming interface) to use the window's current coordinate for the specified parameter.

RESOLUTION

Microsoft has confirmed this to be a bug in the products list above.

This behavior may be a problem for application developers because they may want to set the maximized or minimized horizontal coordinate of a window to -1. To avoid this problem, the developer should trap values of -1, and use a value of -2 or 0 (zero) as appropriate.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbbuglist
KBSubcategory: UsrWndw

BUG: SNMP Service Produces Bad "Error on getproc(InitEx) 127"

PSS ID Number: Q130699

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SYMPTOMS

When the SNMP service is started with debug level 2 or greater, it returns this error message:

```
error on getproc(InitEx) 127
```

RESOLUTION

This error message should be ignored.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

How to Start the SNMP Service

The SNMP service can be started from either the control panel or from a console window.

Type "net help start snmp" in a console box to see how to configure error logging of the SNMP agent.

The syntax of the command is:

```
net start snmp [/logtype: type] [/loglevel: level]
```

where:

- /LOGTYPE: type determines where the log will be created. The possible values are 2 for file, 4 for eventlog, and 6 for both. The default is 4. The file option creates a file under \WINNT\SYSTEM32 called SNMPDBG.LOG.

- /LOGLEVEL: level determines the debug level. The higher the number, the more the detail obtained. The default is 1 (minimum), and the range is

from 1 to 20.

Here is an example:

```
net start snmp /logtype:6 /loglevel:10
```

This starts the SNMP service with loglevel 10, and logs events in the eventlog as well as in SNMPDBG.LOG.

The SNMP service can also be started from a console window without typing "net start." This makes all error messages go to the console window and can be used to help in debugging when writing SNMP applications. For example, the following starts the SNMP service:

```
cmd-prompt> snmp
```

Additional reference words: 3.10 3.50
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkSnmp

BUG: Socket Inheritance in Windows 95 and Windows NT 3.51

PSS ID Number: Q150523

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.51 and Windows 95

SYMPTOMS

Windows 95 does not treat inheritance of Winsock socket handles in the same manner as Windows NT. This article summarises the difference between the two operating systems.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

On Windows NT, socket handles are inheritable by default. This feature is often used by a process that wants to spawn a child process and have the child process interact with the remote application on the other end of the connection.

It is also common practice on Windows NT to set the standard handles (standard input, output, or error) of the child process to the socket handle. In such cases, the child process usually does not know that its standard handles are actually sockets.

Windows 95 differs from Windows NT in the following manner:

- Socket handles are not inheritable when created. To ensure that a child process can obtain and use a socket handle created in the parent, the handle must be explicitly duplicated using the Win32 API DuplicateHandle. Set the bInheritHandle parameter of the API to TRUE.

- Socket handles cannot be set to the standard handles of the child process. A programmer may use other mechanisms to pass the socket handle to the client, such as passing the handle values as command line arguments so that the child process can simply look at its argument vector.

The following segment of code illustrates how to write applications that will inherit sockets in child processes on both Windows 95 and Windows NT. Please note that this is 32-bit code only. 16-bit applications cannot inherit socket handles.

Code Sample

```
   // This is a Winsock server that is listening on a port

   // When a client connects, the server spawns a child process and
   // passes the socket handle to the child.
   // The child can use this socket handle to interact with the
   // client and the parent is free to go back to waiting for
   // other clients to connect.

   OrigSock=accept(listen_socket,(struct sockaddr *)&from,&fromlen);

   if (OrigSock == INVALID_SOCKET)  {
      fprintf(stderr,"accept failed %d\n",GetLastError());
      return -1;
   }
   {
      STARTUPINFO si;
      PROCESS_INFORMATION pi;
      char argbuf[256];

      memset(&si,0,sizeof(si));

      //
      // Duplicate the socket OrigSock to create an inheritable copy.
      //
      if (!DuplicateHandle(GetCurrentProcess(),
            (HANDLE)OrigSock,
            GetCurrentProcess(),
            (HANDLE*)&DuplicateSock,
            0,
            TRUE, // Inheritable
            DUPLICATE_SAME_ACCESS)) {

         fprintf(stderr,"dup error %d\n",GetLastError());
         return -1;
      }
      //
      // Spawn the child process.
      // The first command line argument (argv[1])  is the socket handle
      //

      wsprintf(argbuf,"child.exe %d",DuplicateSock);
      if (!CreateProcess(NULL,argbuf,NULL,NULL,
               TRUE, // inherit handles
               0,NULL,NULL,&si,&pi) ){
         fprintf(stderr,"createprocess failed %d\n",GetLastError());
         return -1;
      }

   }
   //
   // The parent must close both copies of the socket handles.
```

```
   //
   closesocket(OrigSock);
   closesocket(DuplicateSock);
```

The following segment of code illustrate how the newly created process would then extract the socket handle from its command line.

```
   main(int argc, char *argv[]){
      SOCKET Sock;

      /* WSAStartup etc. */
      if (2 == argc){
         Sock = atoi(argv[1]);   // use Sock
      }
   }
```

Additional reference words: 4.00
KBCategory: kbnetwork kbbuglist kbcode
KBSubcategory: NtwkMisc

BUG: Sound Fails to Continue in DirectSound

PSS ID Number: Q139924

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0
- Microsoft Game Software Development Kit (SDK) version 1.0

SYMPTOMS

If a DirectSound application's window is minimized while it is playing a sound, the sound fails to resume when the window is maximized.

Note that the transition from the minimized to the maximized state is the only transition that leads to a problem. All other state transitions behave as expected. For example, if a non-maximized window is minimized and then restored to its former size, the sound resumes properly. Also the problem does not occur if a non-minimized window is maximized.

RESOLUTION

If the maximized window for which the sound failed to restart is reduced to a smaller sized, non-minimized window, the sound will restart.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.00 4.00 Windows 95
KBCategory: kbmm kbsound kbbuglist
KBSubcategory: MMWave

BUG: SpoolFile() Fails in Windows 95

PSS ID Number: Q139011

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

SpoolFile() fails in Windows 95, or it fails to remove the spooled file.

RESOLUTION
==========

SpoolFile() will spool the file if the first two parameters are reversed.

SpoolFile() is documented as:

```
HANDLE SpoolFile(lpszPrinter, lpszPort, lpszJob, lpszFile);
```

But, to make it work, call it this way:

```
HANDLE SpoolFile(lpszPort, lpszPrinter, lpszJob, lpszFile);
```

There is no way to get SpoolFile() to remove the spooled file. The file must be removed by the calling application once the file has been printed.

MORE INFORMATION
================

For information on an alternative to SpoolFile(), please see the following article in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q111010
TITLE     : An Alternative to SpoolFile()
```

For more information on sending printer-specific data to a printer in Win32, please see the following article in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q138594
TITLE     : How to Send Raw Data to a Printer by Using the Win32 API
```

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 SpoolFile Raw RAW.DRV PASSTHROUGH
KBCategory: kbprint kbbuglist

KBSubcategory: GdiPrn

BUG: Timimg Out on recvfrom() Causes Windows 95 to Quit

PSS ID Number: Q138267

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95 version 4.0

SYMPTOMS
========

If a Winsock application is continuosly doing a recvfrom() and has a timeout associated with the call, then after some time Windows 95 will either restart or hang (stop responding) as soon as it receives some data.

RESOLUTION
==========

The application should use select() to determine if the socket is ready for reading before doing a recvfrom().

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

This problem is observed in this situation:

An application opens a UDP or IPX socket. It then binds to INADDR_ANY in case of UDP and to network number 0 and host number 0 in case of IPX. It then uses the setsockopt() call with the SO_RCVTIMEO option to set up a timeout of 1 second.

The application then does a recvfrom() in an infinite loop in such a way so that it can receive a packet from any host. After running for approximately 1000 seconds, if a server sends a UDP or IPX broadcast, either the operating system stops resonding or it restarts.

Note that it might be possible to observe the same behaviour with slight variations in circumstances such as different timeouts.

REFERENCES
==========

For more information, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q137914
TITLE : BUG: Windows 95-Based Winsock App Can't Receive IPX Broadacast

Additional reference words: 4.00
KBCategory: kbnetwork kbtshoot kbbuglist
KBSubcategory: NtwkWinsock

BUG: Tooltip for an Edit Control Is Not Displayed

PSS ID Number: Q149700

The information in this article applies to:

- Microsoft Win32s versions 1.3, 1.3c

SYMPTOMS

When a tooltip is assigned to an edit control, it never or very rarely shows up. As a side effect, when the edit control has focus, the mouse pointer blinks with the speed of the caret.

CAUSE

The tooltips code calculates the position to place the tooltip window, by scanning the mask bitmap of the mouse pointer and finding the lowest point that is not transparent. The beam mouse pointer has a mask bitmap that is completely transparent, so the tooltips code never finds a lowest point and places the tooltip too high. Once the tooltip window appears, it gets a WM_MOUSEMOVE message, because the mouse pointer is inside the tooltip window, and therefore disappears immediately.

RESOLUTION

Replace the edit control mouse pointer's bitmap with a non-transparent bitmap.

STATUS

Microsoft has confirmed this to be a bug in Win32s version 1.30 and later. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.30 1.30c
KBCategory: kbprg kbbuglist
KBSubcategory: w32s

BUG: TrueType Fonts Don't Produce Glyphs in Windows NT 3.51

PSS ID Number: Q139510

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows NT version 3.51

SYMPTOMS

Some TrueType fonts from other versions of Windows or third party font vendors may not produce glyphs on Windows NT 3.51.

For example, if installed in Windows NT version 3.51, the Traditional Arabic font (Trad.ttf) from Arabic Windows 3.11 will not produce glyphs. The font installs, but attempts to view it produce the default character indicating that the character codes do not have a glyph.

RESOLUTION

Microsoft has confirmed this to be a problem in Windows NT version 3.51. This problem was corrected in the WINSRV.DLL file included in the latest Windows NT version 3.51 U.S. Service Pack.

MORE INFORMATION

The bug can be reproduced by viewing a suspect font in the character map application in Windows NT version 3.50 and version 3.51. A problematic font has no characters in any of the character subsets in Windows NT 3.51 while the characters do show up in Windows NT 3.50.

The TrueType fonts most likely to encounter this problem are those that have been constructed for foreign language versions of Windows such as Arabic Windows version 3.11 or 3.1.

Additional reference words: default character far east 3.11 3.10
KBCategory: kbprg kbgraphics kbbuglist kbfixlist
KBSubcategory: GdiFnt GdiTt

BUG: Using WM_SETREDRAW w/ TreeView Control Gives Odd Results

PSS ID Number: Q130611

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS

If a program uses the WM_SETREDRAW message to turn off updating of a TreeView control before adding items, the TreeView control can behave strangely.

For example, if the item being added to the control uses the TVI_FIRST style to insert it to the top of the tree and the top of the tree is scrolled above the top of the visible window, it may be impossible to bring the item into view.

Another possible symptom is that the TreeView control doesn't repaint itself at all. These problems occur only if the program has used the WM_SETREDRAW message to turn off updating the TreeView control.

RESOLUTION

Don't use the WM_SETREDRAW message with the TreeView control while adding items to the control.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 95
KBCategory: kbui kbbuglist
KBSubcategory: UsrCtl

BUG: WAVEHDR.dwBytesRecorded Set to Zero in Win32s

PSS ID Number: Q147430

The information in this article applies to:

- Microsoft Win32s version 1.3

SYMPTOMS
========

When you are using the Multimedia low-level Wave input functions in Win32s for recording purposes, the dwBytesRecorded member of the WAVEHDR structure is set to zero when you handle the MM_WIM_DATA message.

CAUSE
=====

Win32s does not copy the dwBytesRecorded member as it should when handling the MM_WIM_DATA message.

RESOLUTION
==========

You will have to call waveInUnprepareHeader() before you examine the dwBytesRecorded member in the WAVEHDR structure when you are handling the MM_WIM_DATA message. Then you will get the correct result.

Note that the dwBytesRecorded value will be non-zero when you handle the MM_WIM_DATA message the second time or later. This non-zero value is still incorrect as it corresponds to the value of the dwBytesRecored from the previous attempt. Therefore, you should always call waveInUnprepareHeader() to get the correct result from dwBytesRecorded member in the MM_WIM_DATA message. This means you will need to call waveInPrepareHeader() again if recording is continued and the buffer is sent back to the driver.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 1.30 win32s winmm low level wave
KBCategory: kbmm kbbuglist
KBSubcategory: w32s

BUG: Win32 on Windows NT Version 3.51 Bug List - Base

PSS ID Number: Q136432

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51

SUMMARY
=======

This article lists the bugs in the Win32 API implemented on Windows NT version 3.51 that were known at the time the product was released.

MORE INFORMATION
================

- Call CreateProcess() with the command line "myapp" in the directory where Myapp.exe exists. CreateProcess() does not execute myapp.exe if the directory that contains Myapp.exe also contains a subdirectory named Myapp.

- FindFirstFile() returns incorrect file time on FAT.

- GetMailSlotInfo() returns the wrong next message size.

- 32-bit string I/O instructions are not supported.

- NetUserSetInfo() returns NERR_PasswordTooShort instead of returning NERR_PasswordHistConflict during password history conflict.

- NetGroupAdd() returns the wrong error code with invalid group name.

- LogonUser() is not multithread protected.

- Logon to a domain doesn't generate a logon event on the DC.

- The documentation for MsgWaitForMultipleObjects() says that the API returns successfully when either the objects are signalled or the input is available. However, the API behaves as if it requires that the objects are signalled and the input is available.

- CreateProcess() fails to locate .exe file is the current directory is a UNC name.

- SetLastErrorEx() does not raise a RIP_INFO debug event.

- The Characteristics field of the IMAGE_IMPORT_DESCRIPTOR structure is declared incorrectly in Winnt.h. The field is really the RVA of the INT array.

- ASCII characters in the command line of a console application are

converted to ANSI.

- The SourceName and Computername fields of the EVENTLOGRECORD structure are not WCHAR if you call the ANSI version of the API. They should be declared TCHAR.

- WINDEF.H should define CDECL as __cdecl.

- AddAtom does not return an error for an illegal parameter.

- CreateFileMapping() of a device handle returns different error codes from FAT, NTFS, and CDFS. The error on NTFS is ERROR_INVALID_PARAMETER, while the error on FAT and CDFS is ERROR_BAD_EXE_FORMAT.

- InterruptRegister() does not notify of Ctrl+Alt+SysRq.

- Windows NT cannot use or boot from a DblSpace drive.

- Overlapped I/O that extends the file is not asynchronous on NTFS.

- Using file compression causes the mouse to become unresponsive.

- Deleting *.tes deletes file.test, because its short name is file~1.tes.

- Windows NT can only mount up to 26 file systems.

- GetInformationByHandle returns an incorrect link count over the network.

- OpenSCManager() does not work when passed the name of the local machine.

- Service Control Manager will not delay more than 20 seconds PENDING time with SERVICE_PAUSE_PENDING, then error 2186 is generated.

- If a service has a timeout of 30 sec, the value stored in the event log is 30 msec.

- The service controller does not allow multiple logged-on services which share a process.

- Redirected drives created by services not listed by net use.

- QueryServiceObjectSecurity corrupts the handle if passed an invalid pointer for the security descriptor.

Additional reference words:
KBCategory: kbprg kbbuglist
KBSubcategory: BseMisc

BUG: Win32 SDK Version 3.5 Bug List - OLE

PSS ID Number: Q122679

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 3.5

SUMMARY

=======

The following lists the bugs in the Win32 SDK and OLE API that were known when the SDK was released.

MORE INFORMATION

================

- Packager will report that there is not enough memory when you try to package a file which is in use.

- VB can't activate Word object if Word is running in separate VDM.

- MFC 2.5 OLE 2.0 Automation Server fails initialization.

- If a client thread terminates without calling CoUnitialize(), its servers in the same process are not released.

- OLE 2.0 hides application aborts. The server does not receive the exception.

- CoCreateInstance() returns REGDB_E_CLASSNOTREG when the object create fails, instead of the return code from DllGetClassObject() from the inproc server DLL.

- OLE objects inherit only "system" environment variables, not "user" environment variables.

- Users can log on and start an ole app before the OLE service is autostarted.

- Passing CoMarshalInterface() a NULL pUnk (pointer to IUnknown) causes an access violation.

- CoGetClassObject() returns E_OUTOFMEMORY when DllGetClassObject() fails.

- If an OLE Server which dies after registering its class, the container will stall waiting for CoGetClassObject() to succeed.

- Class cache never shrinks - all classes are treated as InUse.

- Default handler causes RPC_E_FAULT exceptions during OnClose.

- ::SetColorScheme() does not validate lpLogPal.

- Messages for server ownerdraw menuitems on menu bar go to the container.

- The server is started if you "Insert" a "Link from File" choosing icon format.

- IsLinkUpToDate() returns S_FALSE after object creation. This can cause containers to run the server twice during creation of the object.

- Setting the link source with IOL::SetSourceMoniker() does not update the presentation cache, even though it does run the server app.

- AddRef() does not marshall count back properly for return value.

- BindToStorage() on non-existent file returns MK_E_INVALIDEXTEN.

- IOleCache::Cache() fails for ICON aspect unless metafile format is used.

- IOleCache2::DiscardCache() does not persist uncaches.

- If an enhanced metafile node exists and IOleCache::Cache is used to cache multiple NULL FORMATETCs, a node is added.

- IViewObject::GetColorSet() after flushing cache should return OLE_E_BLANK, but signals Win32 error 0x8007000e.

- OleDuplicateMedium() does not GlobalUnlock() source METAFILEPICT in the error case.

- Iconic aspect has incorrect colors when played in metafile.

- OLE Cache has a limit of 100 nodes, but accepts more and never returns from the call.

- GetColorSet() for WMF should return LOGPALLETE, which is the union of colors used in the contained CreatePalette() calls. GetColorSet() only returns the first colorset found in this case.

Additional reference words: 3.50
KBCategory: kbprg kbbuglist
KBSubcategory: LeTwoMisc

BUG: Win32 SDK Version 3.5 Bug List - Subsystems & WOW

PSS ID Number: Q122048

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 3.5

SUMMARY

=======

The following is a list of the bugs in the Win32 version 3.5 SDK that were known when Win32 version 3.5 was released. The list is divided into three sections: OS/2 Subsystem, POSIX Subsystem, and Windows on Win32 (WOW).

MORE INFORMATION

================

OS/2 Subsystem

- Extended characters in editors don't work with right-hand Alt key.

- DosRmDir() returns ERROR_PATH_NOT_FOUND, not ERROR_FILE_NOT_FOUND, if passed the name of a directory that does not exist.

- CTRL-X doesn't show with OS/2 Epsilon, although it is received by the application.

- DosSelectDisk() returns error if disk not ready, but not under OS/2.

- DosFindFirst() doesn't find config.sys in the registry.

- Ctrl-C doesn't centre text in Word 5.5 for OS/2.

- Ctrl-C will not cancel Fortran 5.1 setup.

- DosQApptype() returns ERROR_INVALID_EXE_SIGNATURE (191) if pszFileName does not use the correct .EXE format, but OS/2 returns ERROR_BAD_EXE_FORMAT (193).

- DosMove() returns error 80 if the target file exists, but OS/2 returns error 5.

- DosSetFileMode() returns error 0x02 if the file does not exist, but OS/2 returns error 0xCE.

- OS/2 subsystem doesn't load REXXINIT.DLL on startup, as OS/2 does, so OS/2 applications using REXX will fail on the first attempt.

- Flags3 field not preserved between DosDevIOCtl/ASYNC_SETDCB and DosDevIOCtl/ASYNC_GETDCB. Flags3 is always returned with a value of 3.

- Many vio functions return the wrong error code.

- Epsilon not jumping back after matching braces.

- NetHandleGetInfo() returns error 87 at level 2 or level 3.

- DosCreateThread() corrupts 11 words of temporary (under OS/2, only
  2 words are corrupted).

- UUPC OS/2 1.x application tries to open and write to the com port. CTS
  and DTR come up, but RTS never does.

- SQL 4.2 setup fails to configure sort order.

- DosKillProcess() of ancestor with DKP_PROCESSTREE succeeds, but OS/2
  returns ERROR_NOT_DESCENDANT.

- DosCWait() with DCWA_PROCESSTREE returns when the process terminates,
  not when the tree terminates.

- OS/2 CMD.EXE hangs after running Epsilon or PWB.

- Signals not held while resizing data segments with DosReallocSeg().

POSIX Subsystem

- printf() doesn't check to see if writing fails on the first character;
  it tries to write the second character instead of returning an error.

Windows on Win32 (WOW)

- GlobalSize() of handle returned by GetMetaFile() may not match.

- Invalid TEMP environment variable causes problems, whereas Windows 3.1
  will use the root if the TEMP environment variable is invalid.

- GetExitCodeProcess() does not return exit codes for WOW applications.

- Floating point exceptions are not reported to the debugger.

- Segment Load Failure in KRNL386 after using WINDISK.EXE.

- Locking at negative offset fails on WOW with EACCES, but works under
  Windows.

- WIN16 emulator incorrectly handles FBSTP instruction on MIPS, Alpha,
  and x86 with no math coprocessor.

- WOW passes wrong size buffer to spooler when 16-bit application passes
  wrong size structure to WOW.

- WOW uses Windows 3.0 devmode structure, not 3.1 devmode structure,
  which has two extra fields.

- Fast Alt+Tab between WOW apps selects the File menu from one of the applications.

- WM_MENUSELECT wParam and lParam are lost in PostMessage().

- EndDeferWindowPos() returns 0 for success, but returns nonzero for success under Windows.

Additional reference words: 3.50 buglist3.50
KBCategory: kbprg kbbuglist
KBSubcategory: SubSys

BUG: Win32 SDK Version 3.5 Bug List - WinDbg Debugger

PSS ID Number: Q122681

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 3.5

SUMMARY

=======

The following is a list of bugs in the WinDbg debugger that were known at the time of the release of the Win32 SDK version 3.5.

MORE INFORMATION

================

- Enter rgbGlobal, s in watch window (note: rgbGlobal is defined as a character array). The variable is displayed as a character string and an array that can be expanded. When expanding the array, the value of each element is "CAN0026: Error: bad format string."

- REP and REPE are the same prefix codes. REPE is to be used for string comparisons and REP for all other instructions. WinDbg always uses REP.

- Type information defined in a DLL is not available when the current context is another DLL or an EXE.

- Breakpoint message classes do not match class list in SPY.

- C++ expresssion evaluator doesn't handle default function arguments. This is because the compiler does not include them in the debug information.

- Locals window updates on radix change.

- Evaluation of a function with breakpoints returns an incomplete result.

- The Memory Window can't be scrolled up before the starting address.

- Locals window collapses expanded structures on change of scope, such as stepping into a block (not a new function).

- The value of array members cannot be changed.

- The expression evaluator does not handle casting from a class to a primitive data type.

- Remove Last in Quickwatch only works once when multiple items are added to the watch list in a single quickwatch session.

- The return value type is not reported for ?<FuncName>.

- Function evaluation reports "Error: function requires implicit

conversion" for a function taking a structure (not a pointer to a structure).

- Watch window shift-key selection is not consistent: sometimes all characters from the beginning of the expression to the caret position are selected, sometimes 2 characters are selected.

- User DLLs dialog silently discards edits after picking a DLL and changing the radio button from suppress to load.

- Information windows don't maintain color after structure expansion.

- WinDbg disassembles F2 66 F0 F0 AF as "repne lock lock sca" not "repne lock lock scasw".

- Breakpoints may not work correctly in multithreaded apps in areas not protected by critical sections.

- Choosing Stop Debugging and Restart causes memory leak (100K per iteration).

- Combo box in dialog for browsing symbol files is too narrow to show the *.dbg.

- Debug.Watch does not set default watch expression to the selection made in the source window.

- If you set a conditional breakpoint, you step over it with an F10, and the condition is not currently satisfied, the program will run to completion, rather than stepping.

- Long expression (?arg00+arg01+...+arg31) causes debuggee to run to termination.

- ?<function returning near pointer> displays segment.

- Windbg hangs if exited during aedebug start.

- A vararg function evaluation fails on Mips and Alpha.

- Private members may not be evaluatable.

- First Command Window prompt after connecting to target machine for kernel debugging is ">", not "KDx86>", "KDMIPS>", or "KDALPHA>".

- Context expression evaluation of item up the callstack cannot be evaluated and causes CXX0036: Error: bad context {...} specification.

- Alpha: Disassembly of RS, RC, RPCC, FETCH, and FETCH_M instructions displays no operand.

- Help file says "u" command is for unfreezing a thread. The "u" command is for unassemble; it is the "z" command that is used for unfreezing a thread.

- Automatic forward searching not done by breakpoint dialog. Otherwise, when setting a breakpoint on a line that does not contain executable code, the breakpoint is set on the next executable line.

- Automatic forward searching not done when modules are loaded. Otherwise, when setting a breakpoint on a line that does not contain executable code, the breakpoint is set on the next executable line.

- OK button not always active on Set Process dialog.

- Alpha: Large enumerated value not displayed correctly (16-bits instead of 32-bits).

- The Delete button in User DLLs dialog is always active.

- ?Spinlock::Spinlock should display the prototype for the function, but it causes CXX0046: Error: argument list required for member function.

- Flat callstack displayed debugging 16-bit Windows-based application.

- File menu Save_All is not enabled consistently on all platforms.

- Page up/down goes farther than scroll thumb in the Memory Window.

- Page up/down doesn't move scroll thumb in Memory Window.

- Disassembler option "Display Symbols" ignored on Alpha.

- Ppcodes always displayed in lower case in MIPS disasembly, even if "Uppercase symbols and opcodes" is checked.

- Create several workspaces for a single program, choose Delete from the Program menu, and select several of the workspaces. WinDbg locks up when you select OK.

- Deleting the last debugger DLL causes an access violation.

- Bad caret movement when editing Memory Window with ASCII format.

- Calls window not updated if the current thread is changed with the Set Thread dialog. The Calls window is updated if the Command window is used to set the current thread.

- Thread-specific translations of segment registers is not done. The segment register is translated using thread 0's descriptor table.

- When stepping over a function which contains a breakpoint, execution halts, but there is no message indicating that a breakpoint was hit.

- Value of "this" pointer is incorrect in a virtual function in a derived class.

- Based pointers in flat segments are displayed as a 16-bit value, not a 32-bit value. In addition, nothing happens when you click the expansion button.

- WINDBG won't set a breakpoint on code placed in memory and then executed.

- Windbg does not know about all exceptions that can occur while debugging 16-bit code.

- Alpha: CVTxx instructions disassembled with 3 operands, instead of only 2 operands. The first operand is wrong, the second operand would be the correct first operand, and the third operand would be the correct second operand.

- !help <str> reports that there is no help available.

- Set a breakpoint on a function call which spans multiple source lines, but don't set the breakpoint on the last line. Save the information and leave the debugger. When you restart WinDbg with the saved information, WinDbg cannot resolve the breakpoint.

- Alpha: Cannot step through call through a function pointer.

- Commands sxeld and sxdld cause the debugger to stop when a DLL is loaded.

- If there are no symbols loaded, double-clicking a symbol in the call stack produces a disassembly window with a starting address of 0.

- The following context operators cause "CXX0036: Error: bad context {...} specification":

  ```
  ?{,functest.c,functest.exe}count
  ?{,functest.c,}count
  ```

  The following context operators cause "CXX0017: Error: symbol not found":

  ```
  ?{,,functest.exe}count
  ?{,,}count
  ```

- When the current instruction is "cmp dword ptr [esp+18],01", the register window shows a calculation based on [esp], rather than [esp+18].

- WinDbg displays only the first letter of a 'const WCHAR *const' variable. Casting the variable to a WCHAR * in the Watch window works around the problem.

- Run windbg -g cmd.exe and invoke a batch file that repeatedly invokes another command; WinDbg will leak memory.

- x86: f2a6 is disassembled as "repnee cmpsb", not "repne cmpsb",
  f2a7 is disassembled as "repnee cmpsb", not "repne cmpsd",
  f2ae is disassembled as "repnee scasb", not "repne scasb",
  f2af is disassembled as "repnee scasd", not "repne scasd",
  f0a6 is disassembled as "locke cmpsb", not "lock cmpsb",
  f0af is disassembled as "locke scasd", not "lock scasd",

f32ea6 is disassembled as "rep cmpsb", not "repe cmpsb",
f326a7 is disassembled as "rep cmpsd", not "repe cmpsd",
f32ea7 is disassembled as "rep cmpsd", not "repe cmpsd",
f366a7 is disassembled as "rep cmpsw", not "repe cmpsw",
f36665a7 is disassembled as "rep cmpsw", not "repe cmpsw",
f326ae is disassembled as "rep scasb", not "repe scasb",
f365af is disassembled as "rep scasd", not "repe scasd",
f33eaf is disassembled as "rep scasd", not "repe scasd",
f3f0af is disassembled as "rep locke scasw", not "repe lock scasw",
f366af is disassembled as "rep scasw", not "repe scasw",
f36636af is disassembled as "rep scasw", not "repe scasw".

- dc doesn't accept the '&' prefix for an address specifier.

- CXX0004: Error: syntax error on reference to float array. For example, the error is produced by "g .115;?Pf[8], where Pf is declared float Pf[11].

- If you have a DLL built with multiple files with the same name (that live in different source directories), you cannot set a break point in 2nd file with same name.

- Error "CXX0034: Error: types incompatable with operator" accessing members, member functions, and overloaded operators of base classes and virtual base classes or a derived class.

- Alpha: WinDbg doesn't display floating part of a float constant.

- Crash dumps fail because of bad symbol lookup. This breaks !process when kernel debugging as well.

Additional reference words: 3.50
KBCategory: kbtool kbbuglist
KBSubcategory: TlsWindbg

BUG: Win32 SDK Version 3.51 Bug List - GDI

PSS ID Number: Q136438

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51

SUMMARY

=======

This article lists the bugs in the Win32 API implemented in Windows NT version 3.51 that were known at the time the product was released.

MORE INFORMATION

================

- GetBitmapBits( ) returns different values from Windows 3.1.
- PolyBezier() call with certain coordinates hangs the GDI.
- MoveToEx() and LineTo() do not give an error when the delta is greater than 2^27.
- DrawText() ignores a single space if it is followed by "\r\n".
- The insertion pointer is set to a wrong place if Alt+0146 is entered twice in a multiple-line edit box with Arial font.
- Use of Far East fonts causes csrss to grow very large.
- ScaleWindowExtEx() and ScaleViewportExt() do not fail when the denominator parameter is zero.
- CreateDC() returns error code 126 (ERROR_MOD_NOT_FOUND) on a printer with no access right. The call should return error code 5 (ERROR_ACCESS_DENIED).
- SetDIBits() succeeds when HBITMAP parameter is a handle to region, pen, or font.
- GetObject() does not check the cbBuffer parameter for negative values.
- UnrealizeObject() returns TRUE when passed an invalid handle.
- EndDoc() reports success when given an information context (IC).
- glGet() does not return an error when called between glBegin and glEnd.
- EnumPrintProcessors() fails for a remote server.
- GetPrintProcessorDirectory() does not return the UNC path.

- EnumMonitors() fails on a remote server.

- Job Priority can be set without Manage Documents permission.

- StartDocPrinter() sets the wrong error code on failure.

- StartDocPrinter() after StartDocPrinter(), with no EndDocPrinter() in between, does not fail. This is because no state checking is performed.

- WritePrinter() should fail after EndDocPrinter().

- WritePrinter() should fail after AbortPrinter().

- WritePrinter() returns the wrong error code after AddJob().

- SetPrinter does not fail with an invalid level of 1.

- EnumPrinters() returns remote printers when only local printers are requested.

- AddPrintProcessor() fails with pPath set to a path.

- WritePrinter() does not dispatch the last cleanup bytes to printer.

- AddPrinter() does not set an error code if pPrinter is NULL.

- ScheduleJob() does not fail if there is no spool file.

- If you send WritePrinter() a server handle, the wrong error code is returned.

- The wrong error code is returned when AddMonitor() is called with the path to an invalid monitor.

- AddPrinter() succeeds with an unknown data type.

- If you send ScheduleJob() an invalid job ID, the wrong error code is returned.

- JOB_INFO_2 time is never set.

- ScheduleJob() fails on a deleted job with no spool file.

- RemoteOpenPrinter() sometimes does not open printers.

- OpenPrintProcessor() failure results in an application hang.

- Cannot set or get the security descriptor for a pIniJob.

- ClosePrinter() cleanup code doesn't keep states correctly.

- AddPrinter with port == NUL returns ERROR_UNKNOWN_PRINTER.

- EnumPrinters(NULL, PRINTER_ENUM_REMOTE) does not work.

- There is no default printer if AddPrinterConnection() is called.

- Rundown does not clean up StartDocPrinter() print jobs.

- DeletePrinter() returns wrong type.

- JOB_INFO_2 PagesPrinted should match Windows 95 when pages is zero.

- There is no security check against the job during OpenPrinter().

- There is ambiguity during OpenPrinter() for printers with same name.

- Character '4' may be printed at upper-left of PCL seperator page.

- SetJob(), GetJob(), and EnumJobs() do not set or return a security descriptor.

- DeletePrinterConnection() on a share name returns TRUE but does not delete the connection.

- Creating large Perpetua fonts (lfHeight > 1500) can cause the system to fail.

- DCIOpenProvider() fails on some video cards. Not all cards support DCI.

- WinWatchDidStatusChange() and WinWatchGetClipList() can be called only from within the DCIBeginAccess() / DCIEndAccess() pair.

Additional reference words:
KBCategory: kbgraphic kbbuglist
KBSubcategory: GdiMisc

BUG: Win32 SDK Version 3.51 Bug List - Multimedia

PSS ID Number: Q136439

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SUMMARY
=======

This article lists the bugs in the Win32 API implemented in Windows NT version 3.51 that were known at the time the product was released.

MORE INFORMATION
================

- AVISave fails. To reproduce the problem, start aviedit, open a file, use CTRL+I to get information on streams, set the start field to a large number, and try to save the file.

- AVIStreamGetFrame() returns NULL if you take an uncompressed stream, copy it into the clipboard, and paste it into a different 32-bit app.

- When querying for the description of a codec, it will be listed twice if you have both the 32-bit and 16-bit version. Both times, the 32-bit description is displayed.

- MCIAVI32 causes a general protection (GP) fault if playing RLE or Video 1 encoded AVI files without these Codecs loaded.

- Shuffle streams 16-bit causes GP fault in AVIFILE.DLL on MIPS.

- When all ACM Codecs are disabled, hardware formats are not available with acmMetrics.

- SoundBlaster 16 driver doesn't enable volume on speaker input.

- SoundBlaster driver setup does not adjust DMA buffer size correctly.

- After executing the MCI command Play Fullscreen At End Element, Status Position can return the wrong value.

- The MCI command Put Source At inconsistently does not return errors for invalid coordinates.

- MediaPlayer jumps back one frame as Clock.avi completes. MCIAVI may return an incorrect position if it does not use the most granular stream.

- Under certain circumstances, Resume Notify Behavior Element returns incorrect notification codes.

- AVIFileWriteData allows you to write a data chunk that has an ID that

already exists.

- MPlayer does not play clipped videos in Word correctly.

Additional reference words:
KBCategory: kbmm kbbuglist
KBSubcategory: MMMisc

BUG: Win32 SDK Version 3.51 Bug List - Networking

PSS ID Number: Q136437

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) version 3.51

SUMMARY
========

The following is the list of bugs in the Win32 API implemented on Windows NT 3.51 that were known at the time the product was released. This list is divided into 4 sections: LanMan API, RAS, RPC, and TCP/IP.

MORE INFORMATION
================

LanMan API

 - NetWkstaGetInfo returns wrong platform ID (PLATFORM_ID_OS2) for Windows for Workgroups.

 - NetGetDCName with a domain name of "." or "?" returns NERR_NetNotStarted. The expected error is NERR_DCNotFound.

 - NetUseEnum with a preferred buffer size less than 40 bytes returns ERROR_MORE_DATA (234) rather than NERR_BufTooSmall (2134).

 - NetServerEnum with a preferred buffer size less than 40 bytes returns ERROR_MORE_DATA (234) rather than NERR_BufTooSmall (2134).

 - NetScheduleJobEnum with a preferred buffer size less than 40 bytes return ERROR_MORE_DATA (234) rather than NERR_BufTooSmall (2134).

 - NetStatisticsGet remoted to downLevel OS/2 server does not work for the service "LanmanWorkstation." It returns error 50.

 - NetAccessEnum with a NULL base path generates an access violation. Use a null string to indicate that no base path is to be used.

 - The following are documentation errors:

     - NetUserModals supports a level 3 structure.

     - NetGroupSetUsers also supports level 1.

     - NetAccessCheck is not in the header files.

     - NetAuditClear parameter LPWSTR service should be marked as a reserved field; it must be NULL.

     - NetAuditClear parameter LPWSTR Service should be marked as a

reserved field; it must be NULL.

- NetWkstaUserSetInfo level 0 returns error 124 (invalid level), but the docs say that level 0 is supported.

- NetMessageBufferSend returns error 123 (invalid computer name) if the FromName is not NULL or the client computer.

- NetReplImportDirDel generates the following assert on a checked build: assert:\net\svcdlls\repl\common\impconf.c line 121. This occurs when an invalid directory name is used.

- NetReplExportDirDel generates the following assert on a check build: assert:\net\svcdlls\repl\common\expconf.c line 120. This occurs when an invalid directory name is used.

- NetMessageNameEnum returns error NERR_Sucess (0) with a zero-length buffer rather than NERR_BufTooSmall.

- NETBIOS send datagram and broadcast datagram return NRC_SYSTEM when given a buffer size of 32000, not NRC_BUFLEN (invalid buffer length).

- The NETBIOS send datagram is issued with a buffer size of 512 and the receive datagram is issued with a buffer size of 40. The receive datagram returns NRC_GOODRET, not NRC_INCOMP (incomplete message).

- WNetEnumResource returns the wrong buffer size if the buffer is not big enough. WHen the buffer is NULL (size 0), the error returned is ERROR_NO_NETWORK.

- Calling the MNet APIs results in the error "call to undefined dynalink" because WFWNET.DRV does not stub these APIs.

- NetAccessGetInfo returns code 53 (ERROR_BAD_NETPATH) with the local server name.

- NTVDM NetServerEnum2 with a bad domain name should return 2320.

- NTVDM NetShareGetInfo level 2 fails with error 87.

- NTVDM NetShareGetInfo level 3 returns NERR_Success.

- NTVDM NetUserAdd remoted to server with long name fails with error 59 (unexpected network error).

- NetServiceEnum resume handle and prefMaxLen ignored. The API allocates the required size and returns all information.

- NetConfigGet, NetConfigGetAll, and NetConfigSet remoted to Windows NT should fail.

- NetReplExportDirEnum ignores the suggested buffer size.

- NetUseEnum with level 2 returns error 50 (not supported), not 124 (invalid level).

- NetWkstaUserSetInfo with an incorrect level returns error 87 (invalid parameter), not 124 (invalid level).

- MS-DOS NetWkstaSetUID2 does not return 2242 NERR_PASSWORD_EXPIRED when Windows NT forces the user to change passwords.

- NetAccessEnum level 0 returns error 234 (ERROR_MORE_DATA) when remoted to OS/2.

- NetServerEnum resume handle resumes call at start of list. This will cause an infinite loop.

- NetMessageNameEnum returns error 1733 (RPC_S_INVALID_TAG) with an incorrect info level.

- NetMessageNameEnum with a resume handle and a small buffer goes into an infinite loop.

- NetMessageNameEnum total entries is off by one on the first iteration if resume key is used.

- NetMessageNameGetInfo returns error 1733 (RPC_S_INVALID_TAG) with an incorrect info level.

- NetAccessGetUserPerms remoted to OS/2 works corrected, but then NET.EXE commands no longer work.

- NetUserModalsSet with level 1007 fails with error 87 when remoted to OS/2.

- NetMessageBufferSend fails with error 50 when remoted to OS/2.

- NetShareCheck fails with error 2311 (NERR_DeviceNotShared) when remoted to OS/2.

- NetGroupSetInfo with a bad level remote to OS/2 succeeds. It should return 124 (invalid level).

- NetSessionEnum with a bad level remoted to OS/2 succeeds. It should return 124 (invalid level).

- NetUserGetGroups with a bad level remoted to OS/2 succeeds. It should return 124 (invalid level).

- NetSessionGetInfo causes assert \rpcxlate\rxapi\sessget.c line 207 if remoted to OS/2 with a bad info level. If you ignore the assertion, the error return is 2221 (user not found), not 124 (invalid level).

RAS

- RasDial exhibits unexpected behaviour when used simultaneously from two threads of the same process.

- Shutting down RAS server from command line does not warn about connected RAS clients and disconnects them.

- RASETHER.DLL version information says it's RASTAPI.DLL, the TAPI compliance layer.

- If a long entry is connected and RASPHONE is brought up with the phone book containing the long entry, then RASPHONE cannot hang up the connection. The error is 6 (handle is invalid).

- RAS fails to establish more than 100 sessions.

RPC

- When a procedure with the [notify] attribute is called, the server stub must also call the server manager routine proc_notify(). No code is generated for this.

- The MIDL compiler does not generate an error when an implicit handle uses a type that has been defined but is not a generic handle type.

- Specifying [first_is(0, ...), last_is(0, ...), size_is(1, ...)] for the "short (*as)" pointer gives runtime error "invalid array bounds" on the client side.

- The MIDL compiler allows [ptr] on interface pointers.

- RPC: Varying multidimensional arrays are not put on wire correctly.

- [out] ref ptr doesn't work properly in MIDL -Oi mode.

- Sizing comes up short on a struct with a union in MIDL.

- MIDL compiler allows typedef [comm_status, fault_status] .ACF.

- The handle_t * binding parameter is broken in -Os stub.

- Unmarshalling 2D fixed arrays on a Mac client causes an access violation on a Windows NT server.

- [in, out] encapsulates a union with interface pointers.

- MIDL produces "warning MIDL2207: value out of range" for the following statement: const char HexNotationChar = '\xFF';.

- MIDL does not give warning M2180 (CASE_VALUE_OUT_OF_RANGE) for the following code:

```
typedef union _small_union switch(small sm) un
{
   case 0:    short *ps;
   case 1:    long  *pl;
   case 256:  short   s; <<-- out of range for a small
   default:   char   ch;
```

```
    } small_union;
```

- MIDL should not accept type byte in a switch.

- In -Oi mode, if the first parameter is a transmit_as parameter, and the presented type is smaller than 32 bits, then the top of stack pointer passed to the interpreter is incorrect.

- MIDL allows multiple types mapped to the same user_marshal type.

- If the following output switches are given on a non-object interface (where these files aren't generated), a warning should be produced.

```
    /dlldata filename
    /iid filename
    /proxy filename
    /sstub filename
```

- MIDL allows a generic handle with [handle] specified twice.

- The following code causes error MIDL2235: [implicit_handle] references a type which is not a handle : [Interface 'implicit']

```
    // In IDL

    typedef  handle_t  PRIMITIVE;

    // In ACF

    [  implicit_handle( PRIMITIVE  long_binding_handle )
    ]  interface implicit
```

- MIDL accepts byte_count on an [in, out] pointer.

- MIDL should generate STDMETHODIMP instead of __stdcall so that the .h files can be used on the Mac.

- MS-DOS RPC install does not list Windows TCP/IP as an option.

- RpcNsBindingImportNext has a small memory leak.

- RpcEpRegister and RpcEpRegisterNoReplace behave differently from the OSF DCE spec in two ways:

  - When replacing an entry in the endpoint map, the annotation string should replace the existing annotation string.

  - When an annotation string greater than 64 characters wide is supplied, the DCE spec specifies that it is truncated. The functions currently return EPT_S_INVALID_ENTRY.

- RpcBindingServerFromClient with ncalrpc fails with an object uuid.

- In connection oriented RPC, the cancel packet does not have the call_id field set.

- RpcServerRegisterAuthInfo with RPC_C_AUTHN_DEFAULT should load the default provider specified in the registry.

- With an HPUX server, a Windows NT client and protocol ncadg_ip_udp, all calls raise exception RPC_S_SERVER_UNAVAILABLE.

- RpcBindingInqObject() does not take 0 as binding handle.

- RPC cancel support does not work on all platforms and transports. Cancels are supported only with Windows NT clients and transports ncacn_ip_tcp and ncacn_spx.

- Client obtains binding from CDS using RpcNsBindingImport routines and calls RpcNsBindingInqEntryName with parameter RPC_C_NS_SYNTAX_DEFAULT. The routine returns error 1736 (invalid syntax). This occurs only if the DCE CDS locator is used.

- RPC_C_BINDING_MAX_COUNT_DEFAULT is documented, but not supported yet.

- When a fully bound handle that does not represent something registered in with EPT, a NULL object uuid, and a NULL host binding handle are supplied to rpc_ep_unregister, the routine returns rpc_s_ok instead of ept_s_cant_perform.

- Possible Deadlock due to TCP/IP thread waiting forever when server has crashed.

- Auto handles in Win16 DLL cause stranded runtime DLLs.

- When rpc_object_inq_type is passed a null uuid as the object uuid, RPC returns status 1710 (object uuid not found), instead of returning the null type uuid (which is the default mapping for the null object uuid).

- RPC_C_PROFILE_DEFAULT_ELT is defined as 0 in rpcnsi.h, but it is defined as 1 by DCE. As a result, rpc_ns_profile_elt_inq_begin, next, and done do not work with nsid on DCE machines.

- Win16 NetBIOS method of obtaining NIC addresses for UUID generation does not check LANAs greater than 4.

- Win16 NotifyRegister callback is owned by the currently running task, therefore, it is cleaned up at task exit. This causes trouble if multiple RPC apps start up and the first to start is not the last to exit.

- RPC name resolution on slow IPX MS-DOS clients may fail if the Novell server is not available.

- RpcServerInqBindings only returns information for a net card.

TCP/IP

- When resolving a name that is not in the hosts file, if DNS is not

configured, gethostbyname fails to go to netbt if the first interface in the machine can not resolve the name.

- Cannot establish trust when DNS for Windows Name Resolution is on.

- NBTSTAT only reports info for first adapter.

- Some IRPs passed from NetBT to TCP have a stack frame of 1.

- If the Domain name is changed on the DHCP server, it is not picked up by NetBT until the clients are rebooted.

- FIND NAME fails on multihome machines.

- Multicast address not removed on IP_DROP_MEMBERSHIP.

- Multiple proxys on same subnet cause client connect problems.

- When the SNMP service is started with debug level 2 or greater, it prints the error message "error on GetProc(InitEx) 127".

- SetService causes access violation when SrvInfo is unitialized.

- SetService should return -1 for SERVICE_FLAG_HARD with dwOperation SERVICE_ADD_TYPE.

- SetService() should not allow SERVICE_FLAG_HARD|SERVICE_FLAG_DEFER with SERVICE_REGISTER or SERVICE_DEREGISTER.

- SetService should fail when you register the same service again.

- Debugger does not free socket descriptors after application faults and causes the debugger to attach to the running process.

- A race condition occurs when two threads share a socket. One thread closes a socket and the other thread tries to do a recvfrom on the socket. This causes problems the next time a socket is bound to the same UDP port.

Additional reference words:
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkMisc

BUG: Win32 SDK Version 3.51 Bug List - OLE

PSS ID Number: Q136433

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) for Windows NT

SUMMARY
=======

This article lists the bugs in the OLE libraries released with Windows NT version 3.51 that were known at the time the product was released.

MORE INFORMATION
================

 - IMoniker::BindToObject always returns MK_E_INVALIDEEXTENSION if an error occurs during open. It should only associate the file extension if the file is not a .doc file.

 - CreateFileMoniker expands letters to UNC names. If in doing so, the path becomes greater than MAX_PATH, CreateFileMoniker returns E_OUTOFMEMORY.

 - Cannot call CoCreateInstance for first time from the DllEntry. If you need the DLL to create an object, you can either do this by having the .exe call a creation function in your DLL or post a message to a window owned by your DLL.

 - IOleCache::SetData returns OLE_E_BLANK for invalid lIndex, FormatEtc, aspect, and target device. It should return DV_E_LINDEX, DV_E_FORMATETC, DV_E_DVASPECT, and DV_E_TARGETDEVICE, respectively.

 - Calling freeze on a frozen icon aspect with an invalid lIndex returns VIEW_S_ALREADYFROZEN rather than DV_E_INDEXFreeze of Icon aspect returns.

 - IStream::Write of 0 bytes returns E_INVALIDARG from a 16-bit OLE application running in the WOW layer. Zero is a valid parameter to pass to Write.

 - "Alt -" accelerator does not work with MDI windows.

 - IOleObject::IsUpToDate always returns S_FALSE after link track move.

 - The proxy for IEnumVARIANT::Next allows a NULL pceltFetched when celt is greater than 1.

 - If you place a dataobject which only has a TYMED_FILE format using OleSetClipboardData, GetClipboardData returns NULL.

 - The file version of IStream::CopyTo uses Windows NT signed 64-bit math to compute values it needs. Some of the math could overflow 63 bits if

the sizes and/or seek pointers of the streams are 63 bits or more.

- OleCreateFromData does not update the OBJREL moniker of the newly created embedded object.

- OleCreateLinkToFile succeeds for non-existent files. Your code should check for the existance of the file before calling this function.

- OleUIAddVerbMenu is not setting up the single menu item correctly.

- OleUIConvert and OleUIChangeIcon crash on invalid hMetaPict.

- CoFreeUnusedLibraries fails to unload DLLs that are no longer in use.

- The AddControl option, when calling OleUIInsertObject, does not work as expected.

Additional reference words:
KBCategory: kbole kbbuglist
KBSubcategory: LeTwoMisc

BUG: Win32 SDK Version 3.51 Bug List - User

PSS ID Number: Q136435

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51

SUMMARY

This article lists the bugs in the Win32 API implemented in Windows NT version 3.51 that were known at the time the product was released.

MORE INFORMATION

- GetSystemMetrics(SM_SHOWSOUNDS) and SystemParametersInfo(SPI_GETSHOWSOUNDS) do not return the same thing.
- Applications do not get a WM_KILLFOCUS if the focus is taken while the window is being created.
- 16-bit Paste Link does not clear the clipboard correctly.
- DlgDirSelectEx() does not fill in the buffer if wParam is 0 or 1.
- GetTabbedTextExtent(), TabbbedTextOut(), TextOut(), and ExtTextOut() all fail when an incorrect string length is passed in.
- SystemParametersInfo(SPI_SETDESKWALLPAPER) does not update the wallpaper correctly.
- The WH_MSGFILTER hook doesn't get MSGF_NEXTWINDOW when the user presses ALT+TAB.
- Use of NUMLOCK as PF1 for Terminal Emulation programs does not work.
- GetKeyState() and GetKeyboardState() do not update the key state if the queue is not being read.
- If MA_NOACTIVATE is returned from WM_MOUSEACTIVATE, no windows are left active.
- Popup menu can not be dismissed without selecting an item if the owner is not the foreground window.
- GetMenuItemInfo() returns bad information for menu items that are not type string.
- Applications receive one extra WM_TIMER message after a call to KillTimer().

- Edit controls clip the right edge of italic text.

- Hotkeys not checked during journal playback.

- Windows NT positions nonclient scrollbars differently from Windows 95.

- CreateDesktop() with pDevMode != NULL causes the screen to go blank.

- Creating a window with WS_POPUP and WS_CHILD causes a general protection (GP) fault.

- CreateWindowStation() fails with ERROR_NOT_ENOUGH_MEMORY when a security descriptor greater than 64K is passed in.

- AdjustWindowRect() returns a bad value in large font video mode.

Additional reference words:
KBCategory: kbui kbbuglist
KBSubcategory: UsrMisc

BUG: Win32 SDK Version 3.51 Bug List - WinDbg

PSS ID Number: Q136434

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SUMMARY

========

This article lists the bugs in the WinDbg debugger released with the Win32 SDK for Windows NT version 3.51 that were known at the time the product was released.

MORE INFORMATION

================

- If you set a breakpoint in a function that is evaluated by EE, execution stops at the breakpoint and incomplete results are returned. Generally, the result is wrong and you are not notified of the problem.

- The z-order of windows is not correctly restored when a workspace is reloaded. For example, the Registers window that was on top when you exited is not when you restart.

- In cases where the application has source files that have identical names but are located in different directories, Windbg may confuse one with the other during source-level debugging. Therefore, putting breakpoints in a line in one of the files may not work.

- If you use F10 to step over a function that contains a conditional breakpoint and the condition is not met, the program runs to completion.

- When the system is under a heavy load, a hardcoded breakpoint hit may be erroneously reported.

- WinDbg may hang if it is opened using the AeDebug registry entry and closed while the debuggee is still active.

- When you use the toolbar to place a breakpoint on a source line that is not an executable line, the breakpoint is correctly placed on the next line that contains executable code. However, if this is done before the module is loaded, you get an unresolved breakpoint when the module is loaded.

- If you ise the Breakpoints dialog box to set a breakpoint on a source line that is not an exectuable line and the system DLLs were loaded with deferred symbols, the debugger will load them now and you will get a message about an uninstantiated breakpoint. The breakpoint should have been placed on the next line that contains executable code.

- WinDbg does not show a wait pointer while setting a breakpoint.

- When you use the scroll thumb in the Memory Window, there is a limit to how far down the view can be changed. Once the thumb is dragged to the bottom, nothing more can be viewed. However, you can move as far beyond that point as you want by using the PAGE DOWN key.

- The PAGE UP and PAGE DOWN keys do not move the scroll thumb in the Memory Window.

- Narrow window size prevents Memory Window reformatting.

- WinDbg won't set a breakpoint in generated code that is executed.

- If the Command window is scrolled away from the current input line, it cannot be scrolled back. Scrolling always stops about one line too soon.

- WinDbg may display a popup window if there is no disk in the drive.

- The sxeld and sxdld commands are not supported.

- Pressing CTRL+SHIFT+HOME does not select text in the Command window.

- For functions that take a class member as an argument, WinDbg displays CXX0019: Error: bad type cast evaluating member functions.

- WinDbg cannot evaluate functions using string constants.

- WinDbg shows the value of thread-local static local variables as 0.

- Pressing p when using remote WinDbg behaves as if you pressed g.

- A maximized source window doesn't show the path or dirty flag.

- WinDbg source window title text is sometimes clipped when the window size is changed.

- A popup box is presented when WinDbg is run using an input script and the debugee cannot be found.

- The WinDbg expression evaluator truncates long strings.

- The floating-point register displays do not line up in fixed columns.

- The Address field defaults to the next token from the last active menu when you click Address on the View menu.

- Type information defined in a DLL is not available when the current context is another DLL or the .exe file.

- Breakpoint message classes are not all-inclusive.

- You cannot set a breakpoint on a message for which no WM_xxx message exists.

- You cannot set a single breakpoint on multiple messages.

- Message enumeration does not highlight messages belonging to the selected class.

- Several break types listed when adding breakpoints are extraneous.

- Exception number processing should not allow bit 28 to be set.

- Length field in breakpoint dialog box does not accept expressions.

- All common file dialog boxes should use a single saved directory.

- Status bar should display toolbar button description.

- Toolbar is clipped if main window is too narrow.

- C++ EE doesn't handle default function arguments.

- Floating point registers cannot be displayed in HEX format.

- Changing radix in the Debug dialog box from the Options menu highlights all the values as changed.

- Pressing CTRL+S does not pause dump command output.

- Locals window collapses expanded structure when scope changes.

- Hex values are displayed using different case for first level and lower level values.

- The value of array members cannot be changed.

- The File Open dialog box does not save the previous file type.

- There is no help for Source/Asm mode.

- WinDbg does not always automatically scroll when selecting text with the mouse on the Disassembly window while the mouse is below the Disassembly window.

- The expression evaluator does not correctly deal with the possiblity of a cast from a class to a primitive data type.

- The User DLLs dialog box shows single status item for multiple possibilities.

- A full path is needed when selecting a DLL from the Debugger DLLs dialog box, and there is no Browse button to choose the path.

- Continue to Cursor on the Run menu should be Continue to Caret, because the execution stop point is indicated by the caret in the active window, not the position of the mouse pointer.

- The symbol information output from the X command is not sorted in alphabetical order.

- EE does not allow "context not allowed" to be an l-value.

- The LN command displays only public symbols, not line numbers.

- Focus changes momentarily when disassembly view overlaps source view and View.Address is updated.

- Constants in Source Windows that have suffixes are improperly syntax colored. The numeric part is colored, but the suffix is not.

- The return value type is not reported for ?<FuncName>.

- Function evaluation reports "Error: function requires implicit conversion" for a function taking a structure, not a pointer to a structure.

- ?<10-byte real> displays a leading blank.

- ?<10-byte real>,<format modifier> fails with modifiers Lf, Lg, LG, Le, and LE.

- Options tool buttons are enabled when no watch item is selected.

- The Disassembly window does not scroll properly. The current line sometimes does not appear in the window.

- Watch window shift-key selection is not consistent. Sometimes all characters from the beginning of the expression to the caret position are selected, but sometimes only two characters are selected.

- The User DLLs dialog box silently discards edits after picking a DLL and changing the radio button from suppress to load.

- The DS register display is nine bits long for MIPS. It is currently displayed with the minimum number of digits necessary for its value. It should be displayed as a three-digit field padded with zeros.

- When you click Stop Debugging on the Run menu, all values in the watch window (left and right panes) are cleared.

- While you step through a program, the information windows display changes data in red. When a structure is expanded, the changed data selection (red) is lost.

- The Command window does not scroll correctly. Unless the Command window is positioned at the bottom, new messages are not always visible.

- "DW BP" currently uses DS as the default selector. It should use SS.

- Expanding character pointers in the Watch window shows only the first character in the string.

- The Replace dialog box is missing a Find Previous Button.

- The Watch window does not preserve watchpoints between runs in a single

debugging session or between separate debugging sessions.

- The File Open dialog box does not default to the program directory of an open workspace.

- ?<function name> does not display the function prototype.

- fr st fails with MIPS host and x86 target with windbgrm.

- In a stack trace, the function name displayed for a function without a symbol is the return address of the next frame.

- The base expression of expanded structure is added using quick watch, not the selected item.

- Breakpoints may not work correctly in multithreaded applications in areas not protected by critical sections.

- Choosing Stop Debugging and Restart causes a memory leak (100K per iteration).

- The Watch dialog does not set default watch expression to the selection made in the source window.

- A Long expression (?arg00+arg01+...+arg31) causes the debuggee to run to termination.

- ?<function returning near pointer> displays segment.

- When setting up for kernel debugging, the user must enter the target processor architecture in the Kernel Debugging dialog box. If the wrong value is selected, the target machine will crash the first time the user tries to set a breakpoint, or change its context.

- Module load messages lock up WinDbg. While a module is loading, no other input is allowed.

- WinDbg cannot debug ntvdm on the RISC platform.

- The Find dialog box can be reselected while the dialog box "The string xxxx was not found" is still open.

- The Cancel button from the "Ask to save workspace if Modified" dialog box causes WinDbg to exit, instead of return.

- You cannot move the caret in the Dissassembly Window by pressing the UP ARROW and DOWN ARROW keys.

- You cannot set a breakpoint on the first line in the Dissassembly window.

- WinDbg can close prematurely if a breakpoint is ambiguous.

- Data type "const WCHAR *const" displays only the first letter of the string.

- A previously added exception is selected when you attempt to add a new exception in the Exceptions dialog box.

- WinDbg is unable to evaluate an enum expression on PPC.

- WinDbg cannot find symbols for class names on PPC. Therefore WinDbg is unable to set breakpoints on constructors or other member functions.

- When server debugging, WinDbg does not look in the symbol search path set in the User DLLs dialog box. Instead, it looks in the current path.

- The Single Message list does not contain WM_ENTERMENULOOP or WM_EXITMENULOOP.

- An Access Violation occurs if the LM command is used when debugging a 16-bit Windows-based application.

Additional reference words:
KBCategory: kbtool kbbuglist
KBSubcategory: TlsWindbg

BUG: Win32 SDK Version 3.51 Bug List - WOW and Subsystems

PSS ID Number: Q136436

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) version 3.51

SUMMARY
=======

This article lists the bugs in the WOW and the OS/2, POSIX, and Windows on Win32 (WOW) Subsystems that were known at the time the product was released.

MORE INFORMATION
================

This list is divided into two sections: WOW and Subsystems.

WOW

 - Windows NT loses long file names saved by 16-bit Windows-based applications.

 - 16-bit Windows-based applications require DLLs on the local machine.

 - GetExitCodeProcess() does not return exit codes for 16-bit Windows-based applications in WOW.

 - Locking at negative offsets fails in WOW.

 - WOW incorrectly handles the FBSTP instruction.

 - WOW incorrectly handles FISTP DWORD PTR.

 - GetWindowPlacement() returns the wrong length in WINDOWPLACEMENT in WOW.

 - WOW EndDeferWindowPos() does not return a nonzero value for success.

 - You cannot change the number of NetBIOS names available for WOW.

 - GetMessageTime() can return a value greater than GetCurrentTime() returns.

 - Custom focus does not work in WOW.

 - WOW WNetServerBrowseDialog() returns WM_SUCCESS, not WM_CANCEL, when the user clicks the Cancel button.

 - WOW WNetDisconnectDialog() returns error 5 for printers under WOW, even printers connected explicitly with the NET USE command.

- Printing from 16-bit Windows-based applications using old quick linker fails in WOW.

Subsystems

- A remote NetUserAdd from the OS/2 subsystem causes an internal OS/2 subsystem error.

- Calls to OS/2 DosSemClose can fail if another process called DosSemWait on the same semaphore.

- OS/2 queue handles should be enumerated starting from 1, not 0.

- OS/2 DosPeekQueue causes a general protection (GP) fault in the OS/2 subsystem.

- POSIX printf does not handle EINTR.

- POSIX does not handle NTFS_MAX_LINK_COUNT.

- Command prompt is changed by MS-DOS-based applications upon exit.

- INT 21, function 60 returns incorrect UNC handles.

- The mouse pointer is reset with Full Screen MS-DOS-based applications.

- The mouse pointer trapped in the upper-left corner with MS-DOS-based applications.

- VddInstallHook() does not block duplicate mappings.

- Line-by-line scrolling by pressing CTRL+UP and CTRL+DOWN does not work in an MS-DOS-based editor if NUM LOCK is enabled.

Additional reference words:
KBCategory: kbprg kbbuglist
KBSubcategory: Subsys

BUG: Win32s 1.25a Bug List

PSS ID Number: Q130138

The information in this article applies to:

- Microsoft Win32s, version 1.25a

The following is a list of the known bugs in Win32s version 1.25 at the time of its release.

- Incorrect context at EXIT_PROCESS_DEBUG_EVENT.

- Progman gets restored when debugger app exits.

- Using StartDoc() does not produce document from printer.

- EM_GETWORDBREAKPROC return code is incorrect.

- Int 3 cannot be trapped via Structured Exception Handling (SEH) on Win32s.

- Win32s does not open all files in RAW mode, as Windows NT does.

- Cannot do ReadProcessMemory (RPM) on memory that has a hardware breakpoint set on it.

- C run-time functions getdcwd()/getcwd() do not work.

- GetFullPathName() returns the root directory for any drive that is not the current drive.

- PlayMetaFileRecord()/EnumMetaFile() contains incorrect lpHTable.

- Size of memory mapped files is rounded to a whole number of pages, meaning that the size is a multiple of 4096 bytes.

- Functions chdrive() and SetCurrentDirectory() fail on PCNFS network drives.

- GetExitCodeProcess() does not return exit codes for 16-bit Windows-based applications.

- Memory passed to Netbios() must be allocated with GlobalAlloc().

- biSizeImage field of BITMAPINFOHEADER is zero.

- CreateFile() on certain invalid long filesnames closes Windows.

- Only the first CBT hook gets messages.

- Most registry functions return the Windows 3.1 return codes, not the Windows NT return codes.

- GlobalReAlloc(x,y,GMEM_MOVEABLE) returns wrong handle type.

- GetVolumeInformation() fails for Universal Naming Convention (UNC)
  root path.

- ResumeThread while debugging writes to debuggee stack.

- GetShortPathName() doesn't fail with a bad path, as it does on
  Windows NT.

- CreateDirectory()/RemoveDirectory() handle errors differently than on
  Windows NT.

- SetCurrentDirctory() returns different error codes than on Windows NT.

- FindText() leaks memory.

- Win32s doesn't support language files other than default (l_intl.nls).

- spawnl does not pass parameters to an MS-DOS-based application.

- Win32s does not support forwarded exports.

- GetDlgItemInt() only translates numbers <= 32767 (a 16-bit integer).

- Changing system locale in Win32s will not have an effect until Win32s
  is loaded again, unlike on Windows NT.

- Module Management APIs missing ANSI to OEM translation.

- When WS_TABSTOP is passed to CreateWindow(), this forces a
  WS_MAXIMIZEBOX.

- Stubbed API FindFirstFileW() does not return -1 to indicate failure.

- SearchPath() and OpenFile() don't work properly with OEM chars in the
  filename.

- GetSystemInfo() doesn't set correct ProcessorType for the Pentium.

- FormatMessage() doesn't set last error.

- FormatMessage() fails with LANG_NEUTRAL | SUBLANG_DEFAULT, but works
  with LANG_ENGLISH | SUBLANG_ENGLISH_US.

- After calling CreateFile() on a write-protected floppy GetLastError()
  returns 2, instead of 19, as it should.

- VirtualProtect() with anything other than PAGE_NOACCESS, PAGE_READ, OR
  PAGE_READWRITE yields unpredictable page protections.

- COMPAREITEMSTRUCT, DELETEITEMSTRUCT, DRAWITEMSTRUCT, AND
  MEASUREITEMSTRUCT incorrectly sign-extend fields.

- GetWindowTextLength() & GetWindowText() incorrectly sign-extend the

   return value.

 - MoveFile() fails on Windows for Workgroups when the source is remote and
   the destination is local.

Additional reference words: 1.25 1.25a
KBCategory: kbprg kbbuglist
KBSubcategory: W32s

BUG: Win32s 1.3 Bug List

PSS ID Number: Q133026

The information in this article applies to:

- Microsoft Win32s version 1.3

The following is a list of the known bugs in Win32s version 1.3 at the time of its release.

- Incorrect context at EXIT_PROCESS_DEBUG_EVENT.

- Progman gets restored when debugger application exits.

- Using StartDoc() does not produce document from printer.

- EM_GETWORDBREAKPROC return code is incorrect.

- Int 3 cannot be trapped via Structured Exception Handling (SEH) on Win32s.

- Win32s does not open all files in RAW mode, as Windows NT does.

- Cannot do ReadProcessMemory (RPM) on memory that has a hardware breakpoint set on it.

- C run-time functions getdcwd()/getcwd() do not work.

- GetFullPathName() returns the root directory for any drive that is not the current drive.

- PlayMetaFileRecord()/EnumMetaFile() contains incorrect lpHTable.

- Size of memory mapped files is rounded to a whole number of pages, meaning that the size is a multiple of 4096 bytes.

- Functions chdrive() and SetCurrentDirectory() fail on PCNFS network drives.

- GetExitCodeProcess() does not return exit codes for 16-bit Windows-based applications.

- Memory passed to Netbios() must be allocated with GlobalAlloc().

- biSizeImage field of BITMAPINFOHEADER is zero.

- CreateFile() on certain invalid long filesnames closes Windows.

- Only the first CBT hook gets messages.

- Most registry functions return the Windows 3.1 return codes, not the Windows NT return codes.

- GlobalReAlloc(x,y,GMEM_MOVEABLE) returns wrong handle type.

- GetVolumeInformation() fails for Universal Naming Convention (UNC)
  root path.

- ResumeThread while debugging writes to debuggee stack.

- GetShortPathName() doesn't fail with a bad path, as it does on
  Windows NT.

- CreateDirectory()/RemoveDirectory() handle errors differently than they
  do in Windows NT.

- SetCurrentDirctory() returns different error codes than does Windows NT.

- FindText() leaks memory.

- Win32s doesn't support language files other than default (l_intl.nls).

- spawnl does not pass parameters to an MS-DOS-based application.

- Win32s does not support forwarded exports.

- GetDlgItemInt() only translates numbers <= 32767 (a 16-bit integer).

- Changing system locale in Win32s has no effect until Win32s
  is loaded again, unlike in Windows NT.

- Module Management APIs missing ANSI to OEM translation.

- Stubbed API FindFirstFileW() does not return -1 to indicate failure.

- FormatMessage() doesn't set last error.

- FormatMessage() fails with LANG_NEUTRAL | SUBLANG_DEFAULT, but works
  with LANG_ENGLISH | SUBLANG_ENGLISH_US.

- After calling CreateFile() on a write-protected floppy disk,
  GetLastError() returns 2, instead of 19, as it should.

- MEASUREITEMSTRUCT and DRAWITEMSTRUCT do not have itemID or itemData
  fields initialized with owner draw menu items.

- With winhlp32, deleting an annotation in a popup causes an unhandled
  exception. This occurs with Windows NT 3.51 and Windows 95 as well.

- SetEnvironmentVariables() does not handle an empty string, an equal
  sign (=), or foreign lowercase characters in the variable name.

- With the help authoring switched on with winhlp32, if you use
  PopupContext with an invalid context number, the error message displayed
  is "Cannot find the windows.hlp file. Do you want to find it?"

- With the help authoring switched on with winhlp32, if you use
  PopupContext with a file that has an .HLP extension but isn't help file,

the error message displayed is "Cannot find the windows.hlp file. Do you want to find it?"

- With the help authoring switched on with winhlp32, if you use PopupContext with a context string that does not exist, no error message is produced.

- You can still write to a file that was opened with GENERIC_READ.

- When a file is opened a second time, the attributes are not updated even with CREATE_ALWAYS.

- PrintDlg() with a NULL hPrintTemplate and PD_ENABLEPRINTTEMPLATE returns 7 (ERROR_ARENA_TRASHED), not 6 (ERROR_INVALID_HANDLE).

- PrintDlg() does not fail with a NULL hSetupTemplate and the flags PD_ENABLESETUPTEMPLATE | PD_PRINTSETUP. The expected return code is 6 (ERROR_INVALID_HANDLE). Instead, the regular Print Dialog is displayed.

- PrintDlg() succeeds with a NULL hInstance.

- PrintDlg() succeeds with a From value that is bigger than the To value.

- PrintDlg() succeeds with an emptry From value or an emptry To value.

- The winhlp32 Find tab does not paint correctly.

- Winhlp32 does not have context-sensitive help for itself.

- The return value of GetDlgCtrlID() is sign-extended. This causes a problem for IDs greater than 0x7fff.

- Winhlp32 cannot play .AVI files.

- The winhlp32 Find highlight feature doesn't work on the first try. After opening and closing the Find options, the problem goes away.

- When a help file with a .CNT file is brought up from a write-protected floppy disk, there's a system error the first time that the help file is invoked. This does not happen if winhlp32 is invoked from the command line (winhlp32 a:file.hlp) or if the .GID is already created.

- When you attempt to print a secondary window from Word version 6.0, an application error occurs.

- When you open up a different help file, the window title changes to the title of the new help file. When you go back to the original help file, the window title does not change back to the original title.

- When you bring up a help file in File Manager with a .CNT file but no keywords, the Find tab is brought up.

- TAB keys and hot keys do not work properly in property sheet controls unless the application's message loop calls PropSheet_IsDialogMessage().

Additional reference words: 1.30
KBCategory: kbprg kbbuglist
KBSubcategory: W32s

BUG: Win95 CreateEnhMetaFile Returns hDC with Characteristics

PSS ID Number: Q151918

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with: Microsoft Windows 95, version 4.0

SYMPTOMS

In Windows 95, calling CreateEnhMetaFile and specifying a reference printer DC (hdcRef) as the first parameter results in an hDC with characteristics of the default printer, not of the hdcRef specified.

RESOLUTION

To work around this problem, obtain the current ENHMETAHEADER information by calling GetEnhMetafileBits. Modify the appropriate members of the structure to reflect the correct values, and then call SetEnhMetafileBits() to write the modified header information back.

STATUS

Microsoft has confirmed this to be a bug in the products listed at the beginning of this article. We are researching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

This problem can be demonstrated when the resolution of the device specified as hdcRef to CreateEnhMetafile is different from the resolution of the default printer.

Steps to Reproduce Problem

1. From the Control Panel, set the default printer to print at 300 dots-per-inch (dpi).

2. Modify the DEVMODE structure using DocumentProperties to reset this resolution to 600 dpi for your application.

3. Create a DC using CreateDC with the modified 600 dpi DEVMODE structure from step 2, specifying the same driver, device, and port name as the default printer.

4. Create an enhanced metafile using CreateEnhMetaFile, passing this printer DC from step 3 as the reference DC.

5. Call GetDeviceCaps on the hDC returned from step 4, and verify that the LOGPIXELSX and LOGPIXELSY values are set to 300 dpi, which is the resolution of the default printer device, and not of the modified printer DC created in step 3. Any attempt to draw a 2x2 rectangle onto this enhanced metafile results in a 4x4 rectangle due to the 300 dpi resolution, instead of the desired 600 dpi.

6. To work around the problem, call GetEnhMetaFileBits to obtain an ENHMETAHEADER structure. The szlDevice member of this structure specifies the resolution of the device on which the picture was created. Modify this to reflect the 600 dpi change and write the modified information back using SetEnhMetaFileBits.

Additional reference words: 4.00
KBCategory: kbgraphic kbprg kbbuglist
KBSubcategory: GdiMeta

BUG: Win95 GDI Resets Window/ViewportOrg in MM_TEXT Mode

PSS ID Number: Q152236

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95, versions 3.1, 4.0

SYMPTOMS

An application marked as 3.1 will have its window origin and viewport origin reset on Windows 95 when StartPage() is called and the device context (DC) is in the MM_TEXT mapping mode.

This is a problem only in Windows 95 for both 16-bit and 32-bit applications marked as 3.1.

CAUSE

This bug occurs when enhanced metafiles (EMF) are used to print in Windows 95.

In Windows 95, all output to a non-Postsript printer spools as EMFs. While EMF spooling or EMF banding, GDI fails to record the window or viewport origins unless the mapping mode is set to something other than MM_TEXT.

This problem should not occur when printing to Postscript printers, because these printers do not support spooling using EMFs.

RESOLUTION

The best way to work around this behavior is to call StartPage() before changing any DC attributes. Doing this guarantees that the DC attributes do not get reset regardless of the operating system or whether the application is marked 3.1 or 4.0.

STATUS

Microsoft has confirmed this to be a problem in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

When an application prints, Windows 95 directs all GDI output calls for a page to an ENHANCED metafile. When EndPage is called, GDI steps through the

bands on the page and plays the metafile into every band. In Windows 95, GDI fails to record the origin and extent information into the metafile unless the mapping mode is set to something other than MM_TEXT.

In Windows 95, StartPage resets the printer DC attributes for 4.0-marked applications, while 3.1-marked applications do not reset the printer DC attributes until EndPage is called. Windows NT 3.x does not reset printer DC attributes at all during a print job. For more information, please see the following article in the Microsoft Knowledge Base:

   ARTICLE ID: Q125696
   TITLE   : StartPage/EndPage Resets Printer DC Attributes in Windows 95

Additional reference words: 3.10 4.00 SetWindowOrg SetViewportOrg
KBCategory: kbgraphic kbbuglist
KBSubcategory: GdiDC

BUG: Windows 95 Access Violation Error After Disabling CTRL+C

PSS ID Number: Q137379

The information in this article applies to:

- Microsoft Win32 Software Development KIT (SDK)
  versions 3.5, 3.51, 4.0

SYMPTOMS

In a Win32 environment, a console application can be terminated by pressing CTRL+C. To disable CTRL+C input, a console application can call the SetConsoleCtrlHandler(NULL, TRUE) API function.

When this API function is called in Windows NT, CTRL+C is ignored if pressed. However, when it is called in Windows 95, pressing CTRL+C generates an Access Violation error. Similarly, when this API is called in Windows 95, pressing CTRL+BREAK generates an Access Violation error.

RESOLUTION

There are two alternatives for the programmer who wants to disable CTRL+C and avoid generating an Access Violation error:

- Install a console control handler to capture and ignore the CTRL+C keypress:

```
SetConsoleCtrlHandler(MyHandler, TRUE);
BOOL MyHandler(DWORD dwEventType)
{
   if ( dwEvnetType == CTRL_C_EVENT
      return TRUE;   // CTRL+C handle by function
   else
      return FALSE;  // pass to next handler
}
```

-or-

- Disable CTRL+C by disabling the ENABLE_PROCESSED_INPUT console mode. Disabling the ENABLE_PROCESSED_INPUT console mode then reports CTRL+C to the input buffer, not the system.

```
SetConsoleMode( hConsoleHandle,
    (Mode & ~ ENABLE_PROCESSED_INPUT) );
```

STATUS

Microsoft has confirmed this to be a bug in the products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes

available.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbbuglist
KBSubcategory: BseCon

BUG: Windows 95 DDEML Faults with Multiple Threads

PSS ID Number: Q137193

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS

If an application has multiple threads calling the DDEML APIs simulataneously, warnings may be generated under debug Windows. In cases, the application may fault inside DDEML.

CAUSE

The DDEML library has a bug that can cause re-entrancy problems in certain, high stress conditions.

RESOLUTION

The solution is to have the calling application serialize access to DDEML using a critical section.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00
KBCategory: kbui kbbuglist
KBSubcategory: UsrDde

BUG: Windows 95 Does Not Allow Send with MSG_PARTIAL Flag

PSS ID Number: Q139800

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

The Winsock API send() on Windows 95 does not allow the last parameter to be set to MSG_PARTIAL. The call fails with Winsock error 10045 (Operation not supported on socket).

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

Winsock applications on Windows NT that use message-oriented protocols like SPX with SOCK_SEQPACKET may use the send() API with the last parameter specified as MSG_PARTIAL to indicate that the message bit should be respected by the transport.

Windows 95 does not allow this option.

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkMisc

BUG: Windows 95 Doesn't Allow open_osfhandle on Socket Handles

PSS ID Number: Q139801

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

The Winsock API _open_osfhandle() on Windows 95 fails when called with a socket handle as the argument. The call returns -1 with errno set to EINVAL.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

Applications on Windows NT may use _open_osfhandle to get a C-runtime file descriptor from a Windows NT socket handle. This enables the application to use the descriptor in C-runtime I/O operations like _lread(), etc.

Windows 95 does not allow this operation.

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkMisc

BUG: Windows 95 Limits Mailslot Names to 8.3 Naming Convention

PSS ID Number: Q139716

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

When you use Mailslots with Windows 95, the mailslot name is silently limited to the MS-DOS 8.3 naming convention including subdirectories. For example, if a mailslot is created in Windows 95 with this name:

```
\\ComputerName\Mailslot\Test_slot
```

then although CreateMailslot returns success, the actual mailslot name is set to this:

```
\\ComputerName\Mailslot\Test_slo
```

The mailslot file name is truncated to eight characters and the file extension is truncated to three characters. Therefore, when a client running under Windows NT sends data to \\ComputerName\Mailslot\Test_slot, the data is never recieved by the Windows 95 server. This is because the names of the two mailslots do not match.

However, when a Windows 95 client sends data to \\ComputerName\Mailslot\Test_slot, the data is recieved by the server. This is because both computers are running Windows 95, so they both truncate the mailslot name to store the name as:

```
\\ComputerName\Mailslot\Test_slo
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

This problem also occurs if a server is running under Windows NT and the client is running under Windows 95. Because the Windows 95 client truncates the name of the mailslot, the Windows NT server is unable to receive the data. Here are examples of names that will not work:

```
\\ComputerName\mailslot\test_slot.slot
\\ComputerName\mailslot\test_slo.slot
\\ComputerName\mailslot\path1\path2\test_slot.slot
```

Here are examples of names that will work:

```
\\ComputerName\mailslot\test_slo.slt
\\ComputerName\mailslot\test_slo.slo
\\ComputerName\mailslot\path1\path2\test_slt.slt
```

Additional reference words: 4.00 Windows 95 prefix suffix
KBCategory: kbnetwork kbbuglist
KBSubcategory:

BUG: Windows 95 SNMP System Description Is Incorrect

PSS ID Number: Q139461

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

The value of the system.sysDescription variable in the Windows 95 SNMP agent (Internet-II MIB) is incorrectly set to "Microsoft Corp. Chicago Beta."

RESOLUTION

Please ignore this value. Windows 95 is a released product along with the SNMP agent.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

REFERENCES

Windows 95 Resource Kit
Win32 compact disc \Docs\Misc\Progref.rtf

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkSnmp

BUG: Windows 95-Based Winsock App Can't Receive IPX Broadacast

PSS ID Number: Q137914

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

A winsock application is unable to receive IPX broadcasts when running under Windows 95 even though it is able to receive UDP broadcasts.

RESOLUTION

To enable a winsock application to recieve IPX broadcasts, use the setsockopt() call with the SO_BROADCAST flag as in this example:

```
   int sock, optval;
   .
   .
   sock = socket(.....);
   err = setsockopt( sock, SOL_SOCKET, SO_BROADCAST, (char*)&optval,
         sizeof(optval));
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

Note that there is no need to use this option for the case of UDP and specification for winsock 1.1 says that this is needed only for sending broadcasts.

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbtshoot kbbuglist
KBSubcategory: NtwkWinsock

BUG: Windows NT Hangs When PolyBezier Gets Certain Coordinates

PSS ID Number: Q137762

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.51

SYMPTOMS

PolyBezier causes Windows NT version 3.51 to hang (stop responding to input) when certain valid coordinates are passed to it. To get out of it, you have to turn off the computer. For example, the following valid coordinates cause the problem when they are passed to PolyBezier:

```
point[0].x =102;
point[0].y =532;
point[1].x =221;
point[1].y =180;
point[2].x =340;
point[2].y =-172;
point[3].x =459;
point[3].y =-524;
```

Other apparently unrelated coordinates can also cause the problem.

WORKAROUND

To work around this bug, try using the PolyBezier32s() function, which is described in the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q135058
TITLE : SAMPLE: How to Draw Cubic Bezier Curves in Windows and Win32s

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

NOTE: This bug will be fixed in a later release of Windows NT.

Additional reference words: Graphics Crash Hang Curves
KBCategory: kbprg kbbuglist
KBSubcategory: GdiMisc

BUG: WinHelp() Call Fails Using HELP_TCARD Option in Win32s

PSS ID Number: Q149967

The information in this article applies to:

- Microsoft Win32 Application Programming Interface, included with Microsoft Win32s, 1.30c

---

SYMPTOMS

The WinHelp() call fails when using the HELP_TCARD option under Win32s. For example, the following WinHelp call fails and WinHelp does not appear:

```
WinHelp(hWnd, "GENERIC.HLP", HELP_TCARD | HELP_CONTEXT, 99)
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

The user is guided through the steps of a task when an application displays a sequence of instructions using training card help. A training card typically consists of text that explains a particular step and authorable buttons associated with TCard macros that allow the user to tell the application what to do next.

The training card instance of Windows Help is initiated by an application calling the WinHelp function and specifying the HELP_TCARD command in combination with another command such as HELP_CONTEXT. Subsequently, when the user clicks an authorable button in the training card, clicks a hot spot assigned to the TCard macro, or closes the training card, Windows Help notifies the application by sending it a WM_TCARD message.

Additional reference words: 1.30c Win32s Winhlp32 TCARD WM_TCARD
KBCategory: kbtool kbbuglist
KBSubcategory: W32s

BUG: Winsock App Over IPX/SPX Over RAS Fails to Connect

PSS ID Number: Q140019

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5 and 3.51

SYMPTOMS
========

A Windows NT client makes a PPP (point-to-point protocol) RAS (remote access server) connection to a Windows NT RAS server with support for IPX/SPX protocols enabled. A user on a Windows NT RAS client runs a Winsock application that binds to the RAS adapter and listens for connections over SPX or SPXII protocol sequences. Another user on the Windows NT RAS server runs a Winsock application that binds to the RAS adapter on the server and tries to call connect() to connect to the application on the RAS client. A call to connect() in this scenario fails with SOCKET_ERROR and WSAGetLastError() returns WSAENETUNREACH(10051).

RESOLUTION
==========

The RAS server administrator must configure the RAS server so that it assigns different network numbers to all RAS clients. This can be done by clearing the "Assign same network number to all IPX clients" check box. On the Control menu, click Network. Then click RemoteAccessService, click Network, and click IPX.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available

Additional reference words: 3.50 3.51
KBCategory: kbprg kbnetwork kbbuglist
KBSubcategory: NtwkWinsock

BUG: Winsock Sends IP Packets with TTL 0

PSS ID Number: Q138268

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.5, 3.51, 4.0

SYMPTOMS

If an application is using IP multicasting on Windows NT version 3.5 or version 3.51 or on Windows 95, then it is possible to send packets with Time to Live (TTL) set to 0.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

Please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q131978
   TITLE   : Receiving, Sending Multicasts in Windows NT Using WinSock

It is possible to change the TTL for an IP datagram. For example:

```
   int ttl  = 0;
   int sock = socket( .... );

   err = setsockopt( sock, IPPROTO_IP, IP_MULTICAST_TTL, (char *)&ttl,
   sizeof(ttl));
```

However, as per the requirements of RFC 1122, a host must not send an IP datagram with TTL = 0. Here is a quote from the relavent section of RFC 1122:

   3.2.1.7 Time-to-Live: RFC-791 Section 3.2

   A host must not send a datagram with a Time-to-Live (TTL) value of zero.

   A host must not discard a datagram just because it was received with TTL
  less than 2.

REFERENCES

For more information, please see the following references:

- \Docs\Misc\Mcast.txt on the Win32 SDK Compact Disc

- The following article in the Microsoft Knowledge Base:

  ARTICLE-ID: Q131978
  TITLE : Receiving, Sending Multicasts in Windows NT Using WinSock

- RFCs 791 and 1122

Additional reference words: 4.00 3.50 3.51 Windows 95
KBCategory: kbnetwork kbtshoot kbbuglist
KBSubcategory:

BUG: WNetGetUniversalName Fails Under Windows 95

PSS ID Number: Q131416

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

The WNetGetUniversalName function takes a drive-based path for a network resource and obtains a data structure that contains a more universal form of the name. This function always fails with error 1200 when called from a 32-bit application running under Windows 95.

WORKAROUND
==========

The functionality provided by WNetGetUniversalName can be implemented using the Win32 network enumeration functions WNetOpenEnum and WNetEnumResource. Here is an example of how to use these functions to implement similiar functionality:

```
#include <windows.h>
#include <stdio.h>

// Function Name:  GetUniversalName
//
// Parameters:     szUniv  - contains the UNC equivalent of szDrive
//                           upon completion
//
//                 szDrive - contains a drive based path
//
// Return value:   TRUE if successful, otherwise FALSE
//
// Comments:       This function assumes that szDrive contains a
//                 valid drive based path.
//
//                 For simplicity, this code assumes szUniv points
//                 to a buffer large enough to accomodate the UNC
//                 equivalent of szDrive.

BOOL GetUniversalName( char szUniv[], char szDrive[] )
{
   // get the local drive letter
   char chLocal = toupper( szDrive[0] );

   // cursory validation
   if ( chLocal < 'A' || chLocal > 'Z' )
      return FALSE;

   if ( szDrive[1] != ':' || szDrive[2] != '\\' )
      return FALSE;
```

```
   HANDLE hEnum;
   DWORD dwResult = WNetOpenEnum( RESOURCE_CONNECTED, RESOURCETYPE_DISK,
                                  0, NULL, &hEnum );

   if ( dwResult != NO_ERROR )
      return FALSE;

   // request all available entries
   const int    c_cEntries   = 0xFFFFFFFF;
   // start with a reasonable buffer size
   DWORD        cbBuffer     = 50 * sizeof( NETRESOURCE );
   NETRESOURCE *pNetResource = (NETRESOURCE*) malloc( cbBuffer );

   BOOL fResult = FALSE;

   while ( TRUE )
   {
      DWORD dwSize   = cbBuffer,
            cEntries = c_cEntries;

      dwResult = WNetEnumResource( hEnum, &cEntries, pNetResource,
                                   &dwSize );

      if ( dwResult == ERROR_MORE_DATA )
      {
         // the buffer was too small, enlarge
         cbBuffer = dwSize;
         pNetResource = (NETRESOURCE*) realloc( pNetResource, cbBuffer );
         continue;
      }

      if ( dwResult != NO_ERROR )
         goto done;

      // search for the specified drive letter
      for ( int i = 0; i < (int) cEntries; i++ )
         if ( pNetResource[i].lpLocalName &&
              chLocal == toupper(pNetResource[i].lpLocalName[0]) )
         {
            // match
            fResult = TRUE;

            // build a UNC name
            strcpy( szUniv, pNetResource[i].lpRemoteName );
            strcat( szUniv, szDrive + 2 );
            _strupr( szUniv );
            goto done;
         }
   }

done:
   // cleanup
   WNetCloseEnum( hEnum );
   free( pNetResource );
```

```
   return fResult;

}
```

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 4.00 Win95
KBCategory: kbnetwork kbbuglist kbcode
KBSubcategory: NtwkWinnet

BUG: WSARecvEx Fails to Set MSG_PARTIAL Flag on Windows 95

PSS ID Number: Q139799

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

The Winsock API WSARecvEx() on Windows 95 does not set the Flags parameter to MSG_PARTIAL even if the message was not completely received.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are reasearching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

Winsock applications on Windows NT that use message-oriented protocols like SPX with SOCK_SEQPACKET may use the WSARecvEx API instead of recv. The last parameter of the API is set to MSG_PARTIAL if there was not enough buffer space to receive the complete message. This enables an application to detect partial messages.

Windows 95 does not set the flags to MSG_PARTIAL.

REFERENCES
==========

Win32 SDK Online Reference, WSARecvEx Function.

Additional reference words: 4.00 Windows 95
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkMisc

BUG:Invalid Default Interface for IP Multicasting Causes Crash

PSS ID Number: Q132434

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.5 and 3.51

SYMPTOMS

Changing the default interface for IP multicasting if the interface is not valid for the machine causes Windows NT to crash (stop responding) on the Intel x86 platform.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. We are researching this bug and will post new information here in the Microsoft Knowledge Base as it becomes available.

A fix to this problem is in development, but has not been regression-tested and may be destabilizing in production environments. Microsoft does not recommend implementing this fix at this time. Contact Microsoft Product Support Services for more information on the availability of this fix.

MORE INFORMATION

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. A socket option is available to override the default for subsequent transmissions from a given socket. For example

```
unsigned long addr = inet_addr("12.13.14.15");
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, (char *)&addr, sizeof(addr))
```

where "addr" is the local IP address of the desired outgoing interface. An address of INADDR_ANY may be used to revert to the default interface.

NOTE: In the above case although the IP address is a "legal" address, if it does not belong to the machine, the following behavior occurs depending on what platform is being used:

- Windows NT crashes (stops responding) on the Intel x86 platform.
- Windows NT locks up on the MIPS platform.
- No adverse effect occurs on the ALPHA platform.

There is no adverse effect (regardless of the platform) if the machine actually "owns" the IP address.

REFERENCES

==========

For more information on IP multicasting, refer to the
\DOC\MISC\Multcast.txt file included with the Win32 SDK.

Additional reference words: 3.50
KBCategory: kbnetwork kbbuglist
KBSubcategory: NtwkWinsock

Button and Static Control Styles Are Not Inclusive

PSS ID Number: Q74297

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The class-specific window styles for button controls (BS_*) are mutually exclusive, as are the class-specific window styles for static controls (SS_*). In other words, they cannot be OR'd together as can most styles for edit controls, list boxes, and combo boxes.

For example, the following style is invalid:

```
BS_OWNERDRAW | BS_AUTORADIOBUTTON
```

A button control is either owner-draw or it is not. In the same manner, the following style is also invalid:

```
SS_LEFT | SS_GRAYFRAME
```

WINDOWS.H defines button and static styles sequentially (1, 2, 3...), instead of as individual bits (1, 2, 4...) as other control's styles are defined. If sequential styles are OR'd together, the resulting style may be completely different from that intended.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Byte-Ordering in a Data Packet Under NDIS

PSS ID Number: Q89374

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1, 3.5, and 3.51

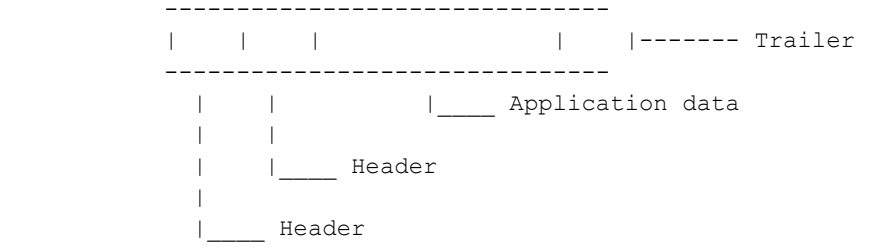
SUMMARY

There may be a difference in the byte order in which 16-bit words are stored in memory versus the order in which the two bytes must be transmitted onto the network as part of a data packet. This difference will depend on the processor involved and which part of the data packet the information falls under.

The information is this article is consistent with the Network Device Interface Specification (NDIS) version 3.0.

MORE INFORMATION

Consider that the data will consist of header(s), the application data, and a trailer(s).



When a protocol driver or an NDIS driver sends information, it does not modify the application data, nor does it modify the ordering of bytes in integer fields.

The ordering of bytes in integer fields within the application data is the responsibility of the Remote Procedure Call (RPC) facility or another mechanism. However, the driver should be concerned with how the information in the header(s)/trailer(s) is stored. For example, the driver may be required to put a 16-bit checksum in a header. To put that integer value into the header in the format required by the network specification, the driver may need to know the type of processor that it is running on and will in any case need to follow the network standard for storing the information in the header.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory:

Calculating String Length in Registry

PSS ID Number: Q94920

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

When writing a string to the registry, you must specify the length of the string, including the terminating null character (\0). A common error is to use strlen() to determine the length of the string, but to forget that strlen() returns only the number of characters in the string, not including the null terminator.

Therefore, the length of the string should be calculated as:

```
strlen( string ) + 1
```

Note that a REG_MULTI_SZ string, which contains multiple null-terminated strings, ends with two (2) null characters, which must be factored into the length of the string. For example, a REG_MULTI_SZ string might resemble the following in memory:

```
string1\0string2\0string3\0laststring\0\0
```

When calculating the length of a REG_MULTI_SZ string, add the length of each of the component strings, as above, and add one for the final terminating null.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

Calculating Text Extents of Bold and Italic Text

PSS ID Number: Q74298

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

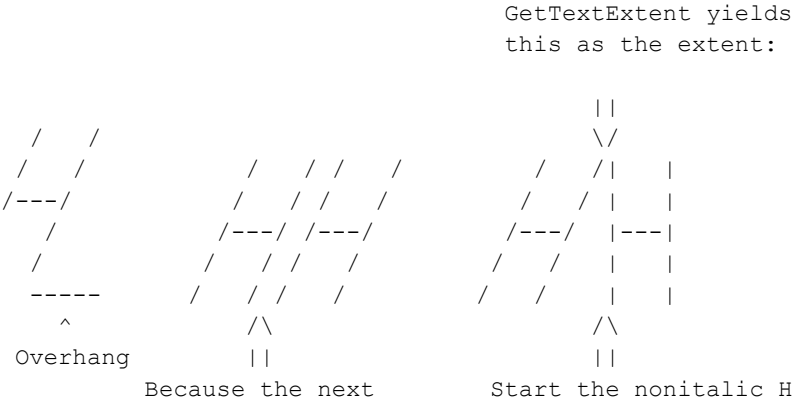
GetTextExtent() can be used to calculate the extent of a string. The value returned may need to be adjusted, depending upon the style of the font. When an italic or bold font is requested and none are available, the graphics device interface (GDI) may simulate those styles using an existing raster or vector font.

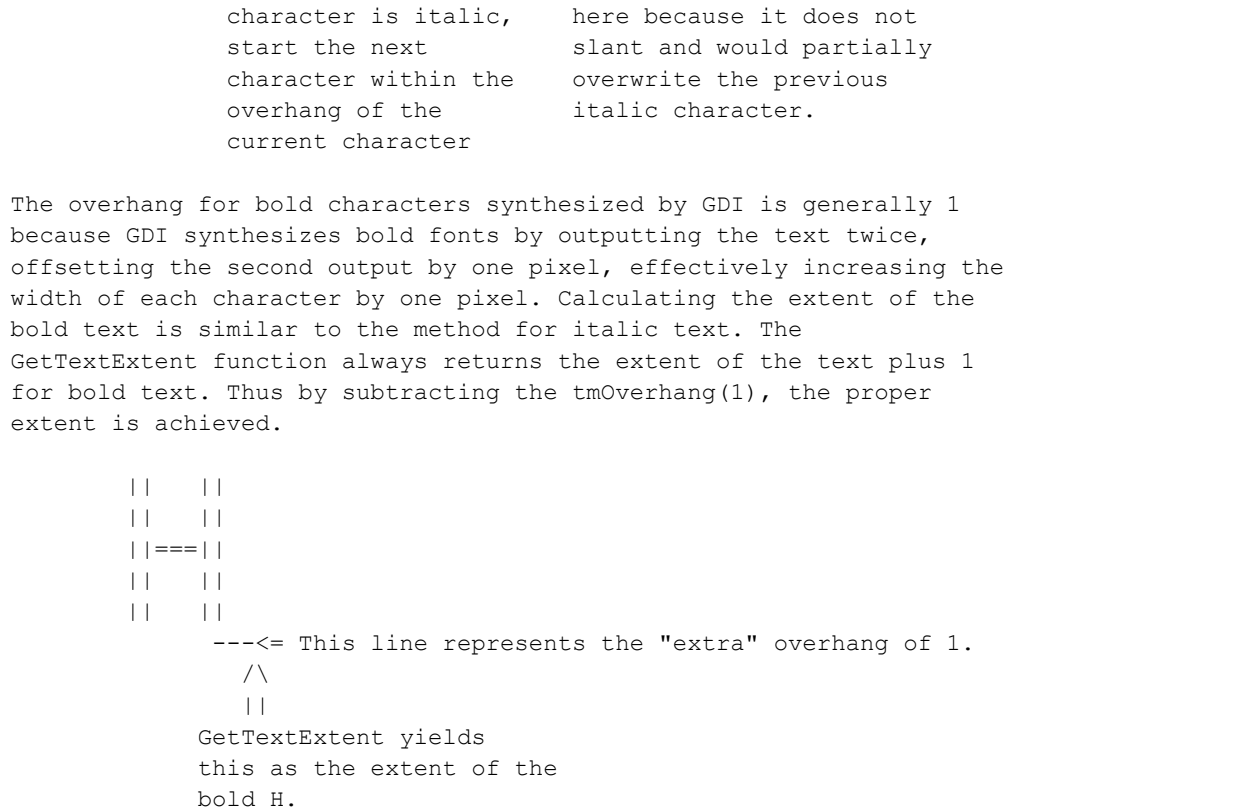
MORE INFORMATION

GDI-simulated bold and italic fonts both include overhangs. The overhang is specified in a TEXTMETRIC structure obtained by calling the GetTextMetrics function. The proper method for calculating the extent of a line of italic or bold text is shown below:

```
dwExtent = GetTextExtent(hDC, lpString, nCount);
GetTextMetrics(hDC, &tm);
xExtent = LOWORD(dwExtent) - tm.tmOverhang;
```

Listed below are examples of italic text alignment. If the next character is not italic, the overhang should not be subtracted from the extent returned from the GetTextExtent function. The overhang needs to be subtracted only when the next character has the same style.





The overhang for bold characters synthesized by GDI is generally 1 because GDI synthesizes bold fonts by outputting the text twice, offsetting the second output by one pixel, effectively increasing the width of each character by one pixel. Calculating the extent of the bold text is similar to the method for italic text. The GetTextExtent function always returns the extent of the text plus 1 for bold text. Thus by subtracting the tmOverhang(1), the proper extent is achieved.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Calculating The Logical Height and Point Size of a Font

PSS ID Number: Q74299

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

To create a font in the Microsoft Windows graphical environment given only the required point size, an application must calculate the logical height of the font because the CreateFont() and CreateFontIndirect() functions use logical units to specify height.

To describe a font to the user, an application can calculate a font's point size, given its height. This article provides the formulas required to perform these calculations for the MM_TEXT mapping mode. You will have to derive a new equation to calculate the font size in another mapping mode.

MORE INFORMATION

To calculate the logical height, use the following formula:

$$\text{height} = \text{Internal Leading} + \frac{\text{Point Size} * \text{LOGPIXELSY}}{72}$$

LOGPIXELSY is the number of pixels contained in a logical inch on the device. This value is obtained by calling the GetDeviceCaps() function with the LOGPIXELSY index. The value 72 is significant because one inch contains 72 points.

The problem with this calculation is that there is no method to determine the internal leading for the font because it has not yet been created. To work around this difficulty, use the following variation of the formula:

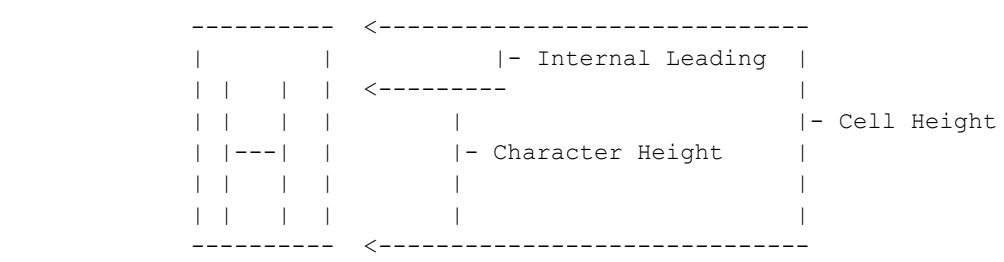
$$\text{height} = \frac{-(\text{Point Size} * \text{LOGPIXELSY})}{72}$$

This formula may also be written as follows:

```
plf->lfHeight = -MulDiv (nPtSize, GetDeviceCaps (hdc, LOGPIXELSY), 72);
```

When an application calls the CreateFont() or CreateFontIndirect()

functions and specifies a negative value for the height parameter, the font mapper provides the closest match for the character height rather than the cell height. The difference between the cell height and the character height is the internal leading, as demonstrated by the following diagram:



The following formula computes the point size of a font:

$$\text{Point Size} = \frac{(\text{Height} - \text{Internal Leading}) * 72}{\text{LOGPIXELSY}}$$

The Height and Internal Leading values are obtained from the TEXTMETRIC data structure. The LOGPIXELSY value is obtained from the GetDeviceCaps function as outlined above.

Round the calculated point size to the nearest integer. The Windows MulDiv() function rounds its result and is an excellent choice to perform the above calculation.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Calculating the Point Size of a Font

PSS ID Number: Q74300

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

The generic formula listed below can be used to compute the point size of a font in the MM_TEXT mapping mode. Any other mapping mode will require a different equation, because the Height will be in a different unit.

MORE INFORMATION

================

$$\text{Point Size} = \frac{(\text{Height} - \text{Internal Leading}) * 72}{\text{LOGPIXELSY}}$$

Height - Cell height obtained from the TEXTMETRIC structure.

Internal Leading - Internal leading obtained from TEXTMETRIC structure.

72 - One point is 1/72 of an inch.

LOGPIXELSY - Number of pixels contained in a logical inch on the device. This value can be obtained by calling the GetDeviceCaps() function and specifying the LOGPIXELSY index.

The value returned from this calculation should be rounded to the nearest integer. The Windows MulDiv() function rounds its result and is an excellent choice for performing the above calculation.

Additional reference words: 3.00 3.10 3.50 4.00 95

KBCategory: kbgraphic

KBSubcategory: GdiFnt

Calculating the TrueType Checksum

PSS ID Number: Q102354

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

To calculate a TrueType checksum:

1. Sum all the ULONGS in the .TTF file, except the checkSumAdjust field (which contains the calculated checksum). Note that TrueType files are big-endian, while Windows and Windows NT are little-endian, so the bytes must be swapped before they are summed.

2. Subtract the result from the magic number 0xb1b0afba.

MORE INFORMATION
================

Example
----

1. Open the SYMBOL.TTF distributed with Windows NT. It is 64492 bytes long.

2. Step through the 16123 ULONGS, summing each one, except for the checkSumAdjust field for the file (which in this case is 0xa7a81151).

3. Subtract the result from 0xb1b0afba. The result is 0xa7a81151.

The TrueType font file specification is available from several sources, including the Microsoft Software Library (query on the word TTSPEC1).

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Call the Windows Help Search Dialog Box from Application

PSS ID Number: Q86268

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

In the Microsoft Windows environment, an application can invoke the Search dialog box of the Windows Help application independent of the main help window. For example, many applications have an item like "Search for Help on" in their Help menus.

An application can invoke the Search dialog box via the WinHelp function by specifying HELP_PARTIALKEY as the value for the fuCommand parameter and by specifying a pointer to an empty string for the dwData parameter. The following code demonstrates how to call the Windows Help Search dialog box from an application:

```
LPSTR lpszDummy,
      lpszHelpFile;

// Allocate memory for strings
lpszDummy = malloc(5);
lpszHelpFile = malloc(MAX_PATH);

// Initialize an empty string
lstrcpy(lpszDummy, "");

// Initialize the help filename
lstrcpy(lpszHelpFile, "c:\\windows\\myhelp.hlp");

// Call WinHelp function
WinHelp(hWnd, lpszHelpFile, HELP_PARTIALKEY, (DWORD)lpszDummy);
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Calling 16-bit Code from Win32-based Apps in Windows 95

PSS ID Number: Q125715

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

As a developer, you may need to access the functionality provided by a 16-bit DLL from your Win32-based applications. This is true particularly when you do not have the source code for the DLL so that you can port it to Win32. This article discusses the mechanism by which Win32-based DLLs can call Windows-based DLLs. The mechanism is called a thunk and the method implemented under Windows 95 is called a flat thunk.

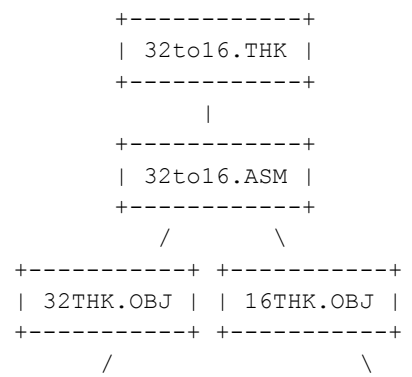
The three major steps in writing the thunk code are:

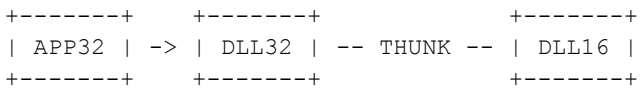
1. Creating the Thunk Script
2. Building the Win32-based DLL
3. Building the Windows-based DLL

MORE INFORMATION

The recommended way to design a thunk call is to isolate all thunk specific code in DLLs (a 16-bit DLL and a 32-bit DLL, to provide both sides of the thunk). That way, you can install certain DLLs on one platform and replace them on another platform, for portability.

Designing a new flat thunk involves creating a thunk script (.THK file). This script is compiled with the Thunk Compiler into an assembly file. This file is assembled using two different flags: -DIS_32 and -DIS_16. This allows you to create both the 16-bit and 32-bit object modules. These object modules are linked in the Windows-based and Win32-based DLLs, respectively. The following diagram summarizes the files involves in building the DLLs.





Creating the Thunk Script

You need to create a script that will be used by the thunk compiler to create a thunk. A thunk script contains the function prototype and a specification for the input and output values. You need to include the following statement to create a 32-bit to 16-bit thunk call:

```
enablemapdirect3216 = true;
```

The following is an example of a simple thunk script for a function that has no input and output:

```
enablemapdirect3216 = true;

void MyThunk16()
{
}
```

The following is an example of script that takes two parameters and returns a value. The second parameter is an output parameter and contains a pointer that is passed back to the Win32-based DLL.

```
enablemapdirect3216 = true;

typedef int   BOOL;
typedef char *LPSTR;

BOOL MyThunk32(LPSTR lpstrInput, LPSTR lpstrOutput)
{
   lpstrOutput = output;
}
```

The following thunk script uses more complex parameter types, such as structures. This example also shows how to specify input and output parameters.

```
enablemapdirect3216 = true;

typedef int BOOL;
typedef unsigned int UINT;
typedef char *LPSTR;

typedef struct tagPOINT {
   INT  x;
   INT  y;
} POINT;
typedef POINT *LPPOINT;

typedef struct tagCIRCLE {
```

```
      POINT center;
      INT   radius;
   } CIRCLE;
   typedef CIRCLE *LPCIRCLE

   void MyThunk32( LPCIRCLE lpCircleInOut)
   {
      lpCircleInOut = InOut;
   }
```

The statement "lpCircleInOut = InOut" tells the script compiler that this pointer is going to be used for input and output.

The thunk compiler usage is as follows:

```
   thunk.exe /options <inputfile> -o <outputfile>
```

The following command line shows how to create a 16-bit thunk code.

```
   thunk -t thk 32to16.thk -o 32to16.asm
```

The "-t thk" option tells the thunk compiler to prefix the thunk functions in the assembly language file with "thk_." This will create an assembly language file.

Building the Win32-based DLL (DLL32)

1. In the DllEntryPoint function (DllMain if you're using the Microsoft C Run-time libraries) in your Win32-based DLL, you must make a call to the imported function thk_ThunkConnect32, as shown here:

```
      BOOL WINAPI DllMain(HINSTANCE hDLLInst,
                          DWORD  fdwReason,
                          LPVOID lpvReserved)
      {
         if (!thk_ThunkConnect32("DLL16.DLL",
                                 "DLL32.DLL",
                                 hDLLInst,
                                 fdwReason))
         {
            return FALSE;
         }
         switch (fdwReason)
         {
            case DLL_PROCESS_ATTACH:
               break;

            case DLL_PROCESS_DETACH:
               break;

            case DLL_THREAD_ATTACH:
               break;

            case DLL_THREAD_DETACH:
```

```
            break;
         }
         return TRUE;
      }
```

2. Include the following lines in the EXPORT section of the module definition (DEF) file for DLL32.

```
   thk_ThunkData32
```

3. Export the function that you are thunking to.

4. Assemble the assembly language file produced by the thunk compiler as a 32-bit object module. The following line shows an example:

```
   ml /DIS_32 /c /W3 /nologo /coff /Fo thk32.obj 32to16.asm
```

5. Link this object module as part of the Win32-based DLL (DLL32).

Building the Windows-based DLL (DLL16)

1. The Windows-based DLL must export a function named "DllEntryPoint". This function must make a call to an imported function thk__ThunkConnect16.

```
   BOOL FAR PASCAL __export DllEntryPoint(DWORD dwReason,
                                          WORD  hInst,
                                          WORD  wDS,
                                          WORD  wHeapSize,
                                          DWORD dwReserved1,
                                          WORD  wReserved 2)
   {
      if (!thk_ThunkConnect16("DLL16.DLL",
                              "DLL32.DLL",
                              hInst,
                              dwReason))
      {
         return FALSE;
      }
      return TRUE;
   }
```

2. Include the following lines in the IMPORTS section of the module definition (DEF) file for DLL16:

```
   C16ThkSL01      = KERNEL.631
   ThunkConnect16  = KERNEL.651
```

3. Include the following lines in the EXPORTS section of the module definition (DEF) file for DLL16. The THK_THUNKDATA16 is defined in the object file that is assembled from the output of the thunk compiler.

```
   THK_THUNKDATA16 @1  RESIDENTNAME
   DllEntryPoint   @2  RESIDENTNAME
```

4. Once you have done that you need to assemble the assembly language file produced by the thunk compiler as a 16-bit object module. The following line shows an example:

   ml /DIS_16 /c /W3 /nologo /Fo thk16.obj 16to32.asm

5. Link this object module as part of the 16-bit DLL (DLL16) object file.

6. Mark the Windows-based DLL as version 4.0. To do this you can use the resource compiler (RC.EXE). The following line shows the syntax:

   rc -40 <DLL file>

   This -40 option is available in the resource compiler that is provided with the Windows 95 SDK and later SDKs.

   NOTE: Be sure to use the RC.EXE in the BINW16 directory so that the application is marked with version 4.0. There is another RC.EXE, but it will not mark the application with version 4.0.

Additional reference words: 4.00 95 flat thunk win16
KBCategory: kbprg
KBSubcategory: SubSys BseMisc

Calling 32-bit Code from 16-bit Apps in Windows 95

PSS ID Number: Q125718

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

It is often desirable to port 16-bit Windows-based applications to Win32 a little at a time, rather than all at once. For example, you may want to port Windows-based DLLs to Win32, but still be able to call them from 16-bit code. This article discusses the mechanism by which Windows-based DLLs can call Win32-based DLLs. The mechanism is called a thunk and the method implemented under Windows 95 is called a flat thunk.

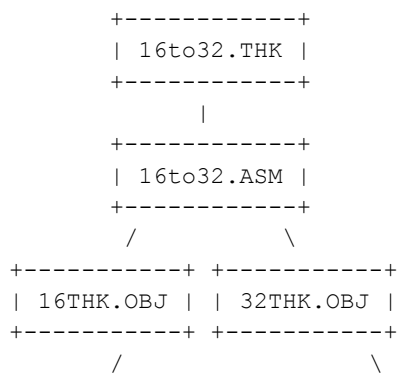
The three major steps in writing the thunk code are:

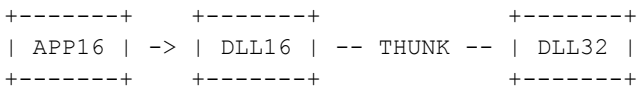
1. Creating the Thunk Script
2. Building the Windows-based DLL
3. Building the Win32-based DLL

MORE INFORMATION

The recommended way to design a thunk call is to isolate all thunk specific code in DLLs (a 16-bit DLL and a 32-bit DLL, to provide both sides of the thunk). That way, you can install certain DLLs on one platform and replace them on another platform, for portability.

Designing a new flat thunk involves creating a thunk script (.THK file). This script is compiled with the Thunk Compiler into an assembly file. This file is assembled using two different flags: -DIS_32 and -DIS_16. This allows you to create both the 16-bit and 32-bit object modules. These object modules are linked in the Windows-based and Win32-based DLLs, respectively. The following diagram summarizes the files involves in building the DLLs.





Creating the Thunk Script

You need to create a script that will be used by the thunk compiler to create a thunk. A thunk script contains the function prototype and a specification for the input and output values. You need to include the following statement to create a 16-bit to 32-bit thunk call:

```
enablemapdirect1632 = true
```

By default, the Win32-based DLL is loaded only on the first encounter of a 16->32 thunk. Because late binding is used, 16-bit code must not depend on any action taken by the initialization of the Win32-based DLL. Also a loading failure of the Win32-based DLL will not be detected until the first 16->32 thunk has been called. To disable late binding of the Win32-based DLL add the following line in your thunk script:

```
preload32=true;
```

The following is an example of a simple thunk script for a function that has no input and output:

```
enablemapdirect1632 = true

void MyThunk32()
{
}
```

The following is an example of script that takes two parameters and returns a value. The second parameter is an output parameter and contains a pointer that is passed back to the Windows-based DLL.

```
enablemapdirect1632 = true

typedef int   BOOL;
typedef char *LPSTR;

BOOL MyThunk32(LPSTR lpstrInput, LPSTR lpstrOutput)
{
   lpstrOutput = output;
}
```

The statement "lpstrOutput = output" tells the script compiler that the 32-bit code will return an address that needs to be converted from flat memory pointer to a selector:offset pointer.

The following thunk script uses more complex parameter types, such as structures. This example also shows how to specify input and output parameters.

```
enablemapdirect1632 = true
```

```
typedef int BOOL;
typedef unsigned int UINT;
typedef char *LPSTR;

typedef struct tagPOINT {
   INT x;
   INT y;
} POINT;
typedef POINT *LPPOINT;

typedef struct tagCIRCLE {
   POINT center;
   INT   radius;
} CIRCLE;
typedef CIRCLE *LPCIRCLE

void MyThunk32( LPCIRCLE lpCircleInOut)
{
   lpCircleInOut = InOut;
}
```

The statement "lpCircleInOut = InOut" tells the script compiler that this pointer is going to be used for input and output. This means that conversion from a 16-bit selector:offset to a flat memory pointer and vice-versa needs to be accomplished.

The thunk compiler usage is as follows:

```
thunk.exe /options <inputfile> -o <outputfile>
```

The following line shows how to create a 16-bit thunk code.

```
thunk -t thk 16to32.thk -o 16to32.asm
```

The "-t thk" option tells the thunk compiler to prefix the thunk functions in the assembly language file with "thk_". This will create an assembly language file.

Building the Windows-based DLL (DLL16)

1. The Windows-based DLL must export a function named "DllEntryPoint". This function must make a call to an imported function thk__ThunkConnect16.

```
BOOL FAR PASCAL __export DllEntryPoint(DWORD dwReason,
                                       WORD  hInst,
                                       WORD  wDS,
                                       WORD  wHeapSize,
                                       DWORD dwReserved1,
                                       WORD  wReserved 2)
{
   if (!thk_ThunkConnect16("DLL16.DLL",
                           "DLL32.DLL",
                           hInst,
```

```
                                dwReason))
         {
            return FALSE;
         }
         return TRUE;
      }
```

2. Include the following lines in the IMPORTS section of the module definition (DEF) file for DLL16.

```
   C16ThkSL01      = KERNEL.631
   ThunkConnect16  = KERNEL.651
```

3. Include the following lines in the EXPORTS section of the module definition (DEF) file for DLL16. The THK_THUNKDATA16 is defined in the object file that is assembled from the output of the thunk compiler.

```
   THK_THUNKDATA16 @1  RESIDENTNAME
   DllEntryPoint   @2  RESIDENTNAME
```

4. Once you have done that you need to assemble the assembly language file produced by the thunk compiler as a 16-bit object module. The following line shows an example:

```
   ml /DIS_16 /c /W3 /nologo /Fo thk16.obj 16to32.asm
```

5. Link this object module as part of the 16-bit DLL (DLL16) object file.

6. Mark the Windows-based DLL as version 4.0. To do this you can use the resource compiler (RC.EXE). The following line shows the syntax:

```
   rc -40 <DLL file>
```

   This -40 option is available in the resource compiler that is provided with the Windows 95 SDK and later SDKs.

   NOTE: Be sure to use the RC.EXE in the BINW16 directory so that the application is marked with version 4.0. There is another RC.EXE, but it will not mark the application with version 4.0.

Building the Win32-based DLL (DLL32)

1. In the DllEntryPoint function (DllMain if you're using the Microsoft C Run-time libraries) in your Win32-based DLL, you must make a call to the imported function thk_ThunkConnect32, as shown here:

```
   BOOL WINAPI DllMain(HINSTANCE hDLLInst,
                       DWORD fdwReason,
                       LPVOID lpvReserved)
   {
      if (!thk_ThunkConnect32("DLL16.DLL",
                              "DLL32.DLL",
                              hDLLInst,
                              fdwReason))
```

```
        {
           return FALSE;
        }
        switch (fdwReason)
        {
           case DLL_PROCESS_ATTACH:
              break;

           case DLL_PROCESS_DETACH:
              break;

           case DLL_THREAD_ATTACH:
              break;

           case DLL_THREAD_DETACH:
              break;
        }
        return TRUE;
     }
```

2. Include the following lines into the EXPORT section of the module definition (DEF) file for DLL32:

   ```
   thk_ThunkData32
   ```

3. Export the function that you are thunking to.

4. Assemble the assembly language file produced by the thunk compiler as a 32-bit object module. The following line shows an example:

   ```
   ml /DIS_32 /c /W3 /nologo /coff /Fo thk32.obj 16to32.asm
   ```

5. Link this object module as part of the Win32-based DLL (Win32).

Additional reference words: 4.00 95 flat thunk win16
KBCategory: kbprg
KBSubcategory: SubSys BseMisc

Calling a New 32-bit API from a 16-bit Application

PSS ID Number: Q125674

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

Windows 95 supports a new set of APIs for 32-bit applications. These APIs are exported by USER32, GDI32, KERNEL32, and so on. In Windows 95, some of these new APIs are also exported by the 16-bit counterpart DLLs in the system such as USER16, GDI16, and so on. But 16-bit applications running on Windows 95 should not call these new APIs from the 16-bit system DLLs because these calls are not unsupported and might be removed from the 16-bit system DLLs in the future.

MORE INFORMATION

APIs such as WindowFromDC(), SetWindowRgn(), SetForeGroundWindow(), and so on for 16-bit USER window management and PolyBezier(), PolyBezierTo(), and so on for 16-bit GDI graphics management are exported from the 16-bit system DLLs.

Even though these APIs are intended for 32-bit applications, the 16-bit system DLLs export some of them. 16-bit Windows 95 Applications should not call them. They are not supported and the APIs will not work as intended.

If Windows 95 Applications need to use these APIs, port the 16-bit application to 32-bit. This is the best solution and is the one that Microsoft recommends. One addtional solution is to write a 32-bit DLL that actually calls the 32-bit API; then the 16-bit application can thunk into this 32-bit DLL. However, Microsoft strongly discourages developers from having applications thunk into sytem DLLs (16- or 32-bit).

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrMisc

Calling a Win32 DLL from a Win16 App on Win32s

PSS ID Number: Q97785

The information in this article applies to:

- Microsoft Win32s versions 1.1 and 1.2

SUMMARY

A Windows version 3.1 application can call a Win32 dynamic-link library (DLL) under Win32s using Universal Thunks.

The following are required components (in addition to the Windows 3.1 application and the Win32 DLL):

- A 16-bit DLL that provides the same entry points as the Win32 DLL. This serves as the 16-bit end for the Universal Thunk. The programmer must also include code that will detect whether the 32-bit side is loaded.

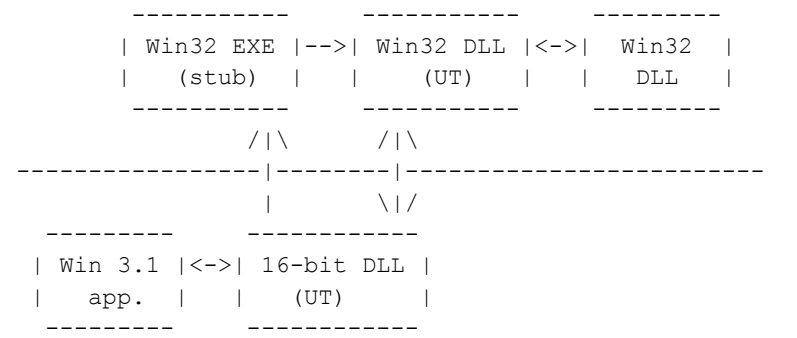
- A Win32 DLL that sets up the Universal Thunk. This serves as the 32-bit end of the Universal Thunk. This DLL is supported only under Win32s.

- A Win32 EXE that loads the 32-bit DLL described above.

NOTE: Universal Thunks were designed to work with a Win32-based application calling a 16-bit DLL. The method described here has limitations. Because the application is 16-bit, no 32-bit context is created, so certain calls will not work from the Win32 DLL. For example, the first time you call malloc() or new() from a DLL entrypoint called by the 16-bit application, the system will hang. This is because MSVCRT20.DLL is using TLS to store C Run-time state information and there is no TLS set up.

MORE INFORMATION

The following diagram illustrates how the pieces fit together:



The load order is as follows: The Windows 3.1 application loads the 16-bit

DLL. The 16-bit DLL checks to see whether the 32-bit side has been initialized. If it has not been initialized, then the DLL spawns the 32-bit EXE (stub), which then loads the 32-bit DLL that sets up the Universal Thunks with the 16-bit DLL. Once all of the components are loaded and initialized, when the Windows 3.x application calls an entry point in the 16-bit DLL, the 16-bit DLL uses the 32-bit Universal Thunk callback to pass the data over to the 32-bit side. Once the call has been received on the 32-bit side, the proper Win32 DLL entry point can be called.

Note that the components labeled Win32 DLL (UT) and Win32 DLL in the diagram above can be contained in the same Win32 DLL. Remember that the code in the Win32 DLL (UT) portion isn't supported under Windows NT, so this code must be special-cased if the DLLs are combined.

For more information, please see the "Win32s Programmer's Reference."

Additional reference words: 1.10 1.20 reverse universal thunk
KBCategory: kbprg
KBSubcategory: W32s

Calling a Win32 DLL from a Win16 Application Under WOW

PSS ID Number: Q104009

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK)

SUMMARY
=======

Under Windows NT and Windows 95, it is possible to call routines in a Win32 dynamic-link library (DLL) from a 16-bit Windows application using an interface called Generic Thunking. This is not to be confused with Win32s Universal Thunks, which provides this functionality under Windows 3.1.

The Generic Thunking interface consists of functions that allow you to load the Win32 DLL, get the address of the DLL routine, call the routine (passing it up to thirty-two 32-bit arguments), convert 16:16 (WOW) addresses to 0:32 addresses (useful if you need to build up a 32-bit structure that contains pointers and pass a pointer to it), call back into the 16-bit application from the Win32 DLL, and free the Win32 DLL.

Generic Thunks are documented in the Win32 SDK documentation, under "Programming Techniques."

NOTE: It is a good idea to test the Win32 DLL by calling it from a Win32-based application before attempting to call it from a 16-bit Windows-based application, because the debugging support is superior in the 32-bit environment.

MORE INFORMATION
================

The basic steps for calling a function through generic thunking are:

 - Call LoadLibraryEx32W() to load the Win32 DLL.
 - Call GetProcAddress32W() to get the address of the DLL routine.
 - Call the DLL routine using CallProc32W() or CallProcEx32W.

CallProc32W() is a Pascal function which was designed to take a variable number of arguments, a Proc address, a mask, and the number of parameters. The mask is used to specify which arguments should be treated as being passed by value and which parameters should be translated from 16:16 pointers to flat pointers. Note that the low-order bit of the mask represents the last parameter, the next lowest bit represents the next to the last parameter, and so forth.

The problem with CallProc32W() is that you cannot create a prototype for it unless you restrict each file so that it only uses calls to functions that contain the same number of parameters. This is a limitation of the Pascal calling convention. Windows NT 3.5 and later supports CallProcEx32W(), which uses the C calling convention to support variable arguments. However, Windows 95 does not fully support CallProcEx32W().

For more information, see the documentation for CallProcEx32W().

Sample Code

The following code fragments can be used as a basis for Generic Thunks.

Assume that the 16-bit Windows-based application is named app16, the Win32 DLL is named dll32, and the following are declared:

```
typedef void (FAR PASCAL *MYPROC)(LPSTR, HANDLE);

DWORD ghLib;
MYPROC hProc;
char FAR *TestString = "Hello there";
```

The DLL routine is defined in dll32.c as follows:

```
void WINAPI MyPrint( LPTSTR lpString, HANDLE hWnd )
{
   ...
}
```

Attempt to load the library in the app16 WinMain():

```
if( NULL == (ghLib = LoadLibraryEx32W( "dll32.dll", NULL, 0 )) ) {
   MessageBox( NULL, "Cannot load DLL32", "App16", MB_OK );
   return 0;
}
```

Attempt to get the address of MyPrint():

```
if( NULL == (hProc = (MYPROC)GetProcAddress32W( ghLib, "MyPrint" ))) {
   MessageBox( hWnd, "Cannot call DLL function", "App16", MB_OK );
   ...
}
```

Although some of the Generic Thunking functions are called in 16-bit code, they need to be provided with 32-bit handles, and they return 32-bit handles. Therefore, before calling CallProcEx32W() and passing it a handle, you must convert the window handle, hWnd, to a 32-bit window handle, hWnd32:

```
hWnd32 = WOWHandle32( hWnd, WOW_TYPE_HWND );
```

Call MyPrint() and pass it TestString and hWnd32 as arguments:

```
CallProcEx32W( 2, 2, hProc, (DWORD) TestString, (DWORD) hWnd32 );
```

Alternatively, you can use CallProc32W() as follows:

```
CallProc32W( (DWORD) TestString, (DWORD) hWnd32, hProc, 2, 2 );
```

A mask of 2 (0x10) is given because we want to pass TestString by reference (WOW translates the pointer) and we want to pass the handle by value.

Free the library right before exiting WinMain():

```
   FreeLibrary32W( ghLib );
```

NOTE: When linking the Windows-based application, you need to put the following statements in the .DEF file, indicating that the functions will be imported from the WOW kernel:

```
   IMPORTS
      kernel.LoadLibraryEx32W
      kernel.FreeLibrary32W
      kernel.GetProcAddress32W
      kernel.GetVDMPointer32W
      kernel.CallProcEx32W
      kernel.CallProc32W
```

The complete sample can be obtained by downloading GTHUNKS.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  - On the www.microsoft.com home page, click the Support icon
  - Click Knowledge Base, and select the product
  - Enter kbfile GTHUNKS.EXE, and click GO!
  - Open the article, and click the button to download the file

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib/Mslfiles folder
  - Get GTHUNKS.EXE

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type "mssupport" (without the quotation marks)
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Locate and Download GTHUNKS.EXE

- Microsoft Download Service (MSDL)
  - Dial (206) 936-6735 to connect to MSDL
  - Download GTHUNKS.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

```
   ARTICLE-ID: Q119591
   TITLE     : How to Obtain Microsoft Support Files from Online Services
```

Faults

On MIPS systems, an alignment fault will occur when a Win32-based application deferences a pointer to unaligned data that was passed by a 16-bit Windows application. As a workaround, declare the parameter with the UNALIGNED keyword. For example,

```
   void func( DWORD *var );
```

becomes

```
   void func( DWORD unaligned *var);
```

An application can use SetErrorMode() to specify SEM_NOALIGMENTFAULTEXCEPT flag. If this is done, the system will automatically fix up alignment faults and make them invisible to the application.

The default value of this error mode is OFF for MIPS, and ON for ALPHA. So on MIPS platforms, an application MUST call SetErrorMode() and specify SEM_NOALIGMENTFAULTEXCEPT if it wants the system to automatically fix alignment faults. This call does not have to be made on ALPHA platforms. This flag has no effect on x86 systems. Note that the fix above is preferable.

Additional reference words: 3.10 3.50 4.00 softlib GTHUNKS.EXE
KBCategory: kbprg kbfile
KBSubcategory: SubSys

Calling a Win32 DLL from an OS/2 Application

PSS ID Number: Q119217

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.5

SUMMARY

=======

The OS/2 subsystem provides a general mechanism that allows 16-bit OS/2 and PM applications to load and call a Win32 DLL. This feature is useful if you would like to call one of the Win32 APIs without having to spawn a Win32 application to do so; it is also useful if you are porting your OS/2 application to Win32 in stages. A small set of APIs (described in the "MORE INFORMATION" section, below) provides this functionality to your application.

To take full advantage of this feature, write a small Win32 DLL to provide the thunk. The OS/2 application calls the thunk DLL, which in turn calls the real Win32 API, using the parameters that were passed from the OS/2 application. You could call the Win32 DLL directly, but the OS/2 subsystem thunking mechanism has only one generic pointer parameter. Most Win32 APIs require different numbers or types of parameters, so the thunking layer is required to retrieve the parameters from the parameter pointer. The parameter pointer typically points to a structure that contains the actual parameters.

You will also need to modify your OS/2 application to call the thunking APIs described below.

MORE INFORMATION

================

The following are the header file, descriptions, and import statements for the 16-bit APIs that are used by the OS/2 application to load and call the thunk DLL. The APIs are defined in the same manner as the OS/2 APIs.

Header File

```
extern USHORT pascal far
Dos32LoadModule(PSZ DllName, PULONG pDllHandle);

extern USHORT pascal far
Dos32GetProcAddr(ULONG Handle, PSZ pszProcName, PULONG pWin32Thunk);

extern USHORT pascal far
Dos32Dispatch(ULONG Win32Thunk, PVOID pArguments, PULONG pRetCode);

extern USHORT pascal far
Dos32FreeModule(ULONG DllHandle);
```

```
   extern USHORT pascal far
   FarPtr2FlatPtr(ULONG FarPtr, PULONG pFlarPtr);

   extern USHORT pascal far
   FlatPtr2FarPtr(ULONG FlatPtr, PULONG pFarPtr);
```

API Descriptions

Dos32LoadModule

```
   USHORT pascal far Dos32LoadModule (
      PSZ    DLLName,
      PULONG pDllHandle );
```

Purpose:

Load the Win32 thunk DLL that will intermediate between the OS/2 application and the Win32 APIs.

Parameters:

DLLName - Name of the thunk DLL.
pDllHandle - Receives the module handle of the DLL. Used as an argument to Dos32GetProcAddr.

Return:

If NO_ERROR is returned, the value pointed to by pDllHandle is used for other WIN32 thunk APIs as described below. It is invalid for usage with regular OS/2 APIs.

If ERROR_MOD_NOT_FOUND is returned, the value pointed to by pDLLHandle is undefined.

Dos32GetProcAddr

```
   USHORT pascal far Dos32GetProcAddr (
      ULONG  DllHandle,
      PSZ    pszProcName,
      PULONG pWin32Thunk );
```

Purpose:

Get a flat pointer to a routine in the Win32 thunk DLL previously opened by Dos32LoadModule.

Parameters:

DllHandle - Handle obtained through Dos32LoadModule.
pszProcName - Name of the API to be called.
pWin32Thunk - Receives pointer to the thunk routine.

Return:

If NO_ERROR is returned, then pszProcName is exported by the thunk DLL.

If ERROR_PROC_NOT_FOUND or ERROR_INVALID_HANDLE is returned, then the value pointed to by pWin32Thunk is undefined.

NOTE: The thunking routine must be declared as _cdecl in order for the Dos32GetProcAddr to work. However, the thunking routine must exit as if it's using a _stdcall in order for Dos32Dispatch to return.

The following program illustrates how the thunking routine should return using the _stdcall convention:

```
 #include <windows.h>
 #include <stdio.h>
 __declspec(dllexport) ULONG Thunk(PVOID pFlatArg);

 __declspec(dllexport) ULONG Thunk(PVOID pFlatArg)
 {
ULONG ulRC = 7;

printf("You have successfully called the Thunk function.\n");
/* put the ULONG return value into the EAX register */
__asm  mov       eax, ulRC
/* now return as this function were declared as _stdcall */
__asm  pop       edi
__asm  pop       esi
__asm  pop       ebx
__asm  leave
__asm  ret       0004

printf("This should be dead code!!!\n");
return (0);
}
```

Dos32Dispatch

```
USHORT pascal far Dos32Dispatch (
   ULONG  Win32Thunk,
   PVOID  pArguments,
   PULONG pRetCode );
```

Purpose:

Call the thunk routine through the pointer obtained through Dos32GetProcAddr.

Parameters:

Win32Thunk - Thunk routine obtained through Dos32GetProcAddr.

pArguments - Argument for thunk routine.
pRetCode   - Error code returned from the thunk routine.

The structure pointed to by pArguments is application specific and
the value is not modified to the OS/2 subsystem.

On the Win32 side (in the thunk DLL), the thunk looks like:

```
   ULONG MyFunc (
      PVOID pFlatArg );
```

The OS/2 subsystem does translate pArguments from a 16:16 pointer to
a flat pointer (pFlatArg). To translate pointers inside the structure,
use FarPtr2FlatPtr.

Return:

If NO_ERROR is returned, then pFlatArg is a valid pointer.

Dos32FreeModule

```
USHORT pascal far Dos32FreeModule (
   ULONG DllHandle );
```

Purpose:

Unload the Win32 thunk DLL that is intermediating between the OS/2
application and the Win32 APIs.

Parameter:

DllHandle - Handle obtained through Dos32LoadModule.

Return:

If NO_ERROR is returned, then DllHandle corresponds to a Win32 DLL
previously loaded by Dos32LoadModule. After the call, DllHandle is no
longer valid. Otherwise, ERROR_INVALID_HANDLE is returned.

FarPtr2FlatPtr

```
USHORT pascal far FarPtr2FlatPtr (
   ULONG  FarPtr,
   PULONG pFlatPtr );
```

Purpose:

Translates a segmented far pointer to a flat pointer.

Parameters:

FarPtr   - Segmented far pointer.
pFlatPtr - Points to flat pointer.

Return:

If NO_ERROR is returned, FarPtr is a valid 16:16 pointer and pFlatPtr contains a valid flat pointer to be used by Win32 code. Otherwise, ERROR_INVALID_PARAMETER is returned and pFlatPtr is undefined.

FlatPtr2FarPtr

```
USHORT pascal far FlatPtr2FarPtr (
   ULONG  FlatPtr,
   PULONG pFarPtr );
```

Purpose:

Translates a flat pointer to a segmented far pointer.

Parameters:

FlatPtr - Flat pointer.
pFarPtr - Points to a segmented far pointer.

Return:

If NO_ERROR is returned, the flat pointer maps to a valid segmented pointer in the 16-bit application's context and pFarPtr contains a valid segmented pointer to be used by 16-bit OS/2 code. Otherwise, ERROR_INVALID_PARAMETER is returned and pFarPtr is undefined.

IMPORTS Section for the Module Definition File

```
IMPORTS
   DOSCALLS.DOS32LOADMODULE
   DOSCALLS.DOS32GETPROCADDR
   DOSCALLS.DOS32DISPATCH
   DOSCALLS.DOS32FREEMODULE
   DOSCALLS.FARPTR2FLATPTR
   DOSCALLS.FLATPTR2FARPTR
```

Additional reference words: 3.50 OS2
KBCategory: kbprg
KBSubcategory: SubSys

Calling CRT Output Routines from a GUI Application

PSS ID Number: Q105305

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

To use C Run-time output routines, such as printf(), from a GUI application, it is necessary to create a console. The Win32 application programming interface (API) AllocConsole() creates the console. The CRT routine setvbuf() removes buffering so that output is visible immediately.

This method works if the GUI application is run from the command line or from File Manager. However, this method does not work if the application is started from the Program Manager or via the "start" command. The following code shows how to work around this problem:

```
   int hCrt;
   FILE *hf;

   AllocConsole();
   hCrt = _open_osfhandle(
              (long) GetStdHandle(STD_OUTPUT_HANDLE),
              _O_TEXT
          );
   hf = _fdopen( hCrt, "w" );
   *stdout = *hf;
   i = setvbuf( stdout, NULL, _IONBF, 0 );
```

This code opens up a new low-level CRT handle to the correct console output handle, associates a new stream with that low-level handle, and replaces stdout with that new stream. This process takes care of functions that use stdout, such as printf(), puts(), and so forth. Use the same procedure for stdin and stderr.

Note that this code does not correct problems with handles 0, 1, and 2. In fact, due to other complications, it is not possible to correct this, and therefore it is necessary to use stream I/O instead of low-level I/O.

MORE INFORMATION

When a GUI application is started with the "start" command, the three standard OS handles STD_INPUT_HANDLE, STD_OUTPUT_HANDLE, and STD_ERROR_HANDLE are all "zeroed out" by the console initialization routines. These three handles are replaced by valid values when the GUI application calls AllocConsole(). Therefore, once this is done, calling

GetStdHandle() will always return valid handle values. The problem is that the CRT has already completed initialization before your application gets a chance to call AllocConsole(); the three low I/O handles 0, 1, and 2 have already been set up to use the original zeroed out OS handles, so all CRT I/O is sent to invalid OS handles and CRT output does not appear in the console. Use the workaround described above to eliminate this problem.

In the case of starting the GUI application from the command line without the "start" command, the standard OS handles are NOT correctly zeroed out, but are incorrectly inherited from CMD.EXE. When the application's CRT initializes, the three low I/O handles 0, 1, and 2 are initialized to use the three handle numbers that the application inherits from CMD.EXE. When the application calls AllocConsole(), the console initialization routines attempt to replace what the console initialization believes to be invalid standard OS handle values with valid handle values from the new console. By coincidence, because the console initialization routines tend to give out the same three values for the standard OS handles, the console initilization will replace the standard OS handle values with the same values that were there before--the ones inherited from CMD.EXE. Therefore, CRT I/O works in this case.

It is important to realize that the ability to use CRT routines from a GUI application run from the command line was not by design so this may not work in future versions of Windows NT or Windows. In a future version, you may need the workaround not just for applications started on the command line with "start <application name>", but also for applications started on the command line with "application name".

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

Calling DdePostAdvise() from XTYP_ADVREQ

PSS ID Number: Q102571

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The documentation for DdePostAdvise() in the Windows 3.1 Software Development Kit "Programmer's Reference, Volume 2: Functions" states the following in the Comments section:

> If a server calls DdePostAdvise() with a topic/item/format name set that includes the set currently being handled in an XTYP_ADVREQ callback, a stack overflow may result.

MORE INFORMATION
================

This is merely a warning against calling DdePostAdvise() from within a DDE callback function's XTYP_ADVREQ transaction, because it may result in a stack overflow.

Like window procedures, DDE callbacks must be coded with care to avoid infinite recursion (eventually resulting in a stack overflow). Because DdePostAdvise() causes DDEML to send an XTYP_ADVREQ transaction to the calling application's DDE callback function, calling DdePostAdvise() on the same topic/item/format name set as the one currently being handled results in an infinite loop.

An analogous piece of code that has become a classic problem in Windows programming involves calling UpdateWindow() in a WM_PAINT case:

```
   case WM_PAINT:
       InvalidateRect (hWnd, NULL, TRUE);
       UpdateWindow (hWnd);
```

Calling UpdateWindow() as in the code above causes a WM_PAINT message to be sent to a window procedure, and thus results in the same type of infinite recursion that occurs when calling DdePostAdvise() from an XTYP_ADVREQ transaction.

An example of a situation that would lend itself to this scenario would be one where data needs to be updated as a result of a previous data change. There are two ways to work around the stack overflow

problem in this case:

- Post a user-defined message and handle the data change asynchronously. For example,

```
// in DdeCallback:
case XTYP_ADVREQ:
          if ((!DdeCmpStringHandles (hsz1, ghszTopic)) &&
              (!DdeCmpStringHandles (hsz2, ghszItem)) &&
              (fmt == CF_SOMEFORMAT))
            {
                  HDDEDATA  hData;

                  hData = DdeCreateDataHandle ();
                  PostMessage (hWnd, WM_DATACHANGED,hData,);
                  return (hData);
            }
            break;

 // in MainWndProc():
 case WM_DATACHANGED:
         DdePostAdvise (idInst, ghszTopic, ghszItem);
           :
```

- Return CBR_BLOCK from the XTYP_ADVREQ and let DDEML suspend further transactions on that conversation, while the server prepares data asynchronously.

More information on how returning CBR_BLOCK allows an application to process data "asynchronously" may be derived from Section 5.8.6 of the Windows 3.1 Software Development Kit (SDK) "Programmer's Reference, Volume 1: Overview," or by querying on the following words in the Microsoft Knowledge Base:

   DDEML and CBR_BLOCK

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Calling LoadLibrary() on a 16-bit DLL

PSS ID Number: Q123731

The information in this article applies to:

- Microsoft Win32s versions 1.15a and 1.2

SUMMARY
=======

In Win32-based applications, LoadLibrary() returns an HINSTANCE and GetLastError() is used to determine the error. If HINSTANCE is NULL, the DLL was not successfully loaded. If the HINSTANCE is not null, the DLL was loaded and the usage count was incremented; however, you may still see that the last error was set if the DLL is a 16-bit DLL.

NOTE: At this point, the DLL is loaded and the usage count is incremented. Call FreeLibrary() to unload the DLL.

MORE INFORMATION
================

In order to see all possible error returns, you'll need to call SetLastError(0) before calling LoadLibrary(). If HINSTANCE is not NULL and GetLastError() is ERROR_BAD_EXE_FORMAT, the DLL is a 16-bit DLL. You can access the DLL resources and/or printer APIs from your Win32-based application.

To call routines in the 16-bit DLL, you should load and call the DLL via the Universal Thunk. This increments the usage count again. Later, you can use FreeLibrary() to free the DLL from the 16-bit code, but this won't unload the DLL from memory unless you already called FreeLibrary() from the 32-bit code. This is because the usage count is not zero. We recommend you call FreeLibrary() from the 32-bit code after the DLL is loaded by the 16-bit code, so the DLL isn't unloaded and then reloaded.

REFERENCES
==========

For more information on how to get resources from a 16-bit DLL, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q105761
   TITLE   : Getting Resources from 16-bit DLLs Under Win32s

For more information on Universal Thunk, please see Chapter 4 of the Win32s Programmer's Reference.

Additional reference words: 1.20
KBCategory: kbprg
KBSubcategory: W32s

Cancelling Overlapped I/O

PSS ID Number: Q90368

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

There is no routine in the Win32 API to cancel an asynchronous request once it has been issued. When a thread does an overlapped I/O (that is, a write), the system starts up another thread to do the I/O and leaves your thread free to do other work. Once it is started, there is no way to stop it.

If it necessary to interrupt the I/O, you can either

- Split the writes into batches and check for interruptions. For example, you could break a 20 megabyte (MB) write into 20, 1 MB writes.

  -or-

- Create another thread yourself to handle the I/O. Terminating the thread will cancel the I/O. You should have a thread in the process explicitly close the handle to the device.

  -or-

- Close the handle to the device with the pending I/O. The close has the net effect of cancelling the I/O.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseFileio

Cancelling WaitCommEvent() with SetCommMask()

PSS ID Number: Q105302

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

If a serial port is in nonoverlapped mode (without FILE_FLAG_OVERLAPPED) and SetCommMask() is called, the call does not return until any pending WaitCommEvent() calls return. This apparently contradicts the following statement from the SetCommMask() Help

> If SetCommMask() is called for a communications resource while a wait is pending for that resource, WaitCommEvent() returns an error.

and the following statement from the WaitCommEvent() Help:

> If a process attempts to change the device handle's event mask by using the SetCommMask() function while a WaitCommEvent() operation is in progress, WaitCommEvent() returns immediately.

However, this is the expected behavior. If you open a serial port in the nonoverlapped mode, then you can do only one thing at a time with the serial port. SetCommMask() must block while the WaitCommEvent() call is blocking.

If the serial port was opened with FILE_FLAG_OVERLAPPED, WaitCommEvent() will return after SetCommEvent() has been called.

The SDK 3.51 docs have been corrected.

Additional reference words: 3.10 3.50 com1 com2
KBCategory: kbprg
KBSubcategory: BseCommapi

Cannot Load <exe> Because NTVDM Is Already Running

PSS ID Number: Q103863

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

WinDbg can debug 16-bit Windows-based applications running on Windows NT, under the Win16 VDM (virtual MS-DOS machine), NTVDM. By default, each 16-bit Windows-based application run as a thread in NTVDM. On Windows NT 3.5, each application can be run in a separate address space.

If NTVDM is running when the debugger tries to start the application, you get the following error message:

```
Cannot load <exe> because NTVDM is already running
```

Under Windows NT 3.5, to work around this problem, go to the WinDbg Options menu, choose Debug, and check Separate WOW VDM, to allow the debuggee to be run in a different address space.

Alternatively, you could terminate NTVDM. On Windows NT 3.1, your only choice is to terminate NTVDM, because separate address spaces for 16-bit Windows-based application are not supported.

To terminate NTVDM, run PView, select NTVDM, and choose "Kill Process." Note that there may be two NTVDM processes. The one that you want to terminate has one thread for each Windows-based application (plus a few more).

The Windows NT WinLogon is set up to automatically start WoWExec, which automatically starts the Win16 VDM. This behavior can be changed by removing WoWExec from:

```
HKEY_LOCAL_MACHINE\
          Software\
         Microsoft\
        Windows NT\
    CurrentVersion\
          Winlogon\
             Shell
```

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsWindbg

Captions for Dialog List Boxes

PSS ID Number: Q24646

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

To place text into the caption bar specified for a list box, use the SetWindowText() function. First, use GetDlgItem() to get the handle of the list box, then call SetWindowText() to set the list box caption.

MORE INFORMATION

The following code fragment illustrates the necessary steps. Note: The list box includes the WS_CAPTION window style.

```
   ...
   BOOL FAR PASCAL TemplateDlg(hWndDlg, message, wParam, lParam)
   ...
       switch (message)
        {
       case WM_INITDIALOG:
           ...
           /* The following line sets the Listbox caption */
           SetWindowText( GetDlgItem(hWndDlg, IDDLISTBOX),
                         (LPSTR)"Caption");
           for (i = 0; i < CSTR; i++) {
               LoadString(hInstTemplate, IDSSTR1+i, (LPSTR)szWaters, 12);
               SendDlgItemMessage(hWndDlg, IDDLISTBOX, LB_ADDSTRING, 0,
                   (LONG)(LPSTR)szWaters);
               }
           SendDlgItemMessage(hWndDlg, IDDLISTBOX, LB_SETCURSEL, iSel, 0L);
           ...
           return TRUE;

       case WM_COMMAND:
   ...
```


Additional reference words: 3.00 3.10 3.50 4.00 95 listbox
KBCategory: kbui
KBSubcategory: UsrCtl

Caret Position & Line Numbers in Multiline Edit Controls

PSS ID Number: Q68572

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

This article explains how to determine the position (row and column) of the caret and the line number of the first visible line of text in a multiline edit control.

MORE INFORMATION
================

Edit controls process several messages that return information relevant to the position of the caret within the control. These messages help an application determine the line number of the caret relative to the number of lines of text in the control.

Once the line number is known, the application can compute the caret's character position within that line and the line number of the first visible line of text in the control.

An edit control must be subclassed in order to track the caret position because the position changes with mouse clicks and keystrokes. The subclass procedure must process the WM_KEYDOWN and WM_LBUTTONDOWN messages, and compute the caret position upon receipt of each message.

The remainder of this article describes three procedures:

- Finding the line number of the caret position
- Finding the column number of the caret position
- Finding the line number of the first visible line

Note that you may replace any mention of the SendMessage API in this article with the SendDlgItemMessage function. Also note that the term return value refers to the value returned by the SendMessage or the SendDlgItemMessage function.

Finding the Line Number of the Caret Position

Perform the following two steps:

1. Send the EM_GETSEL message to the edit control. The high-order word

of the return value is the character position of the caret relative to the first character in the control.

2. Send the EM_LINEFROMCHAR message to the edit control and specify the value returned from step 1 as wParam. Add 1 to the return value to get the line number of the caret position because Windows numbers the lines starting at zero.

Finding the Column Number of the Caret Position

Perform the following three steps:

1. Send the EM_GETSEL message to the edit control. The high-order word of the return value is the character position of the caret relative to the first character in the control.

2. Send the EM_LINEINDEX message with wParam set to -1. The value returned is the count of characters that precede the first character in the line containing the caret.

3. Subtract the value returned in step 2 from the value in step 1 and add 1 because Windows numbers the columns starting at zero. This result is the column number of the caret position.

Finding the Line Number of the First Visible Line

Windows 3.1 and later define the EM_GETFIRSTVISIBLELINE message, which an application can send to a single line or a multiline edit control. For single line edit controls, this value returned for the message is the offset of the first visible character. For multiline edit controls, the value returned is the number of the first visible line.

Under Windows 95, it would be more efficient to use a combination of GetCaretPos() and EM_CHARFROMPOS.

If an application must be compatible with Windows 3.0, it can perform the following 10-step procedure:

1. Follow steps 1 and 2 of "Finding the Line Number of the Caret Position," presented above, and save the line number.

2. Call the GetCaretPos function to fill a POINT structure with the caret's coordinates relative to the client area of the edit control. (The client area is inside the border.)

3. Call the GetDC function using the handle to the edit control to retrieve a handle to a device context for the edit control. Store this handle in a variable named hDC.

4. Send the WM_GETFONT message to the edit control. The return value is a handle to the font used by the edit control. If the value returned is NULL, proceed to step 6 because the control is using the system default font.

5. Call the SelectObject function to select the font used by the edit control into hDC. Do not call the SelectObject function if WM_GETFONT returned NULL in step 4. Save the value returned by SelectObject in the hOldFont variable.

6. Call the GetTextMetrics function with hDC to fill a TEXTMETRIC data structure with information about the font used by the edit control (the font which is selected into hDC). The field of interest is tmHeight.

7. While the vertical coordinate of the caret is greater than the value of tmHeight, subtract tmHeight from the vertical coordinate and subtract 1 from the line number of the caret from step 1.

8. Repeat step 7 until the vertical coordinate of the caret is less than or equal to tmHeight.

9. Call SelectObject to select hOldFont back into hDC. Then call ReleaseDC to return the display context to the system.

10. The value remaining in the line number variable is the line number of the first visible line in the edit control.

Additional reference words: 3.00 3.10 3.50 4.00 95 caretpos
KBCategory: kbui
KBSubcategory: UsrCrt

Case Sensitivity in Atoms

PSS ID Number: Q38901

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

If the same string is added to an atom table twice, but a different case is used, the string is only stored once; only the first one is present.

The "Microsoft Windows 2.0 Software Development Kit Update" for versions 2.03 and 2.1 states that atoms are case insensitive.

This means that when the AddAtom() function is used, the case is ignored when atoms are compared. Therefore, if AddAtom("DIR") is called, and then AddAtom("dir"), the single atom "DIR" (with a reference count of 2) will result. If AddAtom("dir") is called first, the single atom "dir" will result.

Similarly, the other atom-handling functions are case insensitive. For example, calling FindAtom("dIr") will find the atom "dir", "DIR", or "Dir", and so on.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Cases Where "Normal" Window Position, Size Not Available

PSS ID Number: Q68583

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In Windows version 3.0, the "normal" size and position of a window is not available when that window is maximized (zoomed) or minimized (represented as an icon). In Windows version 3.1, two new functions named GeWindowPlacement and SetWindowPlacement have been added which provide access to normal position information.

MORE INFORMATION

The remainder of this article provides two possible ways to work around this limitation in Windows version 3.0:

1. If the normal size is needed only as the application is shut down, restore the window and retrieve its position before closing the application. The following call can be used to restore the window whose window handle is hWnd:

   ```
   SendMessage(hWnd, WM_SYSCOMMAND, SC_RESTORE, 0L);
   ```

   The GetWindowRect or GetClientRect functions can then be used to obtain the window's position.

2. If the normal size is needed at all times, keep track of the position every time the window receives a WM_MOVE message. If the IsIconic and IsZoomed functions both return FALSE, assume the window is normal and update the position values. Otherwise, do not change the saved position information.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Centering a Dialog Box on the Screen

PSS ID Number: Q74798

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51

When an application developed for the Microsoft Windows graphical environment displays a dialog box, centering the dialog box on the screen is sometimes desirable. However, on systems with high-resolution displays, the application displaying the dialog box may be nowhere near the center of the screen. In these cases, it is preferable to place the dialog near the application requesting input.

To center a dialog box on the screen before it is visible, add the following lines to the processing of the WM_INITDIALOG message:

```
    {
    RECT rc;

    GetWindowRect(hDlg, &rc);

    SetWindowPos(hDlg, NULL,
       ((GetSystemMetrics(SM_CXSCREEN) - (rc.right - rc.left)) / 2),
       ((GetSystemMetrics(SM_CYSCREEN) - (rc.bottom - rc.top)) / 2),
       0, 0, SWP_NOSIZE | SWP_NOACTIVATE);
    }
```

This code centers the dialog horizontally and vertically.

Under Windows 95, you should use the new style DS_CENTER to get the same effect.

Additional reference words: 3.00 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrDlgs

Chaining Parent PSP Environment Variables

PSS ID Number: Q96209

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

Some MS-DOS-based applications change the environment variables of their parent application by chaining through the program segment prefix (PSP). With Windows NT, this functionality doesn't work if the parent is a Win32-based application.

MORE INFORMATION

================

When an MS-DOS-based application is started from a single command shell (SCS), the application inherits a new copy of the environment variables. Any attempts by the MS-DOS-based application to modify its parent's environment variables will not work. When the MS-DOS-based application exits, the SCS will be "restored" to its original state. If another MS-DOS-based application is started, the second application will receive the same environment that the first MS-DOS-based application received.

If an MS-DOS-based application (B) is spawned by another MS-DOS-based application (A), any modifications to application A's environment variables will be reflected when application B exits.

For more information on how environment variables are set, please see the following article(s) in the Microsoft Knowledge Base:

ARTICLE-ID: Q100843
TITLE : Environment Variables in Windows NT

Additional reference words: 3.10
KBCategory: kbprg
KBSubcategory: BseMisc

Changes to the MSTest WFndWndC()

PSS ID Number: Q108227

The information in this article applies to:

- Microsoft Win32 SDK, versions 3.1 and 3.5

A change made to MS-TEST WFndWnd() and WFndWndC() may cause them not to work as they did previously. Specifically, treatment of the first parameter of WFndWnd() and WFndWndC() has changed.

Previously, "" and NULL resulted in the caption of the window being ignored. Now, "" means the window must have an empty caption and NULL means to ignore the caption altogether.

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsMisc

Changing a List Box from Single-Column to Multicolumn

PSS ID Number: Q68580

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

A list box cannot be changed from single column to multicolumn by altering the list box's style bits "on the fly."

The effect of switching a single-column list box to multicolumn can be achieved by creating one single column and one multicolumn list box. Initially, hide the multicolumn list box. To switch the list boxes, hide the single-column list box and show the multicolumn list box.

MORE INFORMATION
================

In general, programmatically changing the style bits of a window usually leads to unstable results. The method of switching between two windows (hiding one and showing the other) can safely switch window styles.

Additional reference words: 3.00 3.10 3.50 4.00 95 listbox
KBCategory: kbui
KBSubcategory: UsrCtl

Changing How Pop-Up Menus Respond to Mouse Actions

PSS ID Number: Q65256

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The TrackPopupMenu function allows an application to receive input from a menu that is displayed anywhere within the application's client area. This article demonstrates how to change the menu's default behavior for mouse selections.

MORE INFORMATION
================

The default action for floating pop-up menus maintained with TrackPopupMenu is as follows:

1. If the menu is displayed in response to a keystroke, the pop-up menu is visible until the user selects a menu item or presses ESC.

2. If the menu is displayed in response to a WM_*BUTTONUP message, it acts as if it were displayed in response to a keystroke.

NOTE: In the context of this article, the * in the WM_*BUTTONUP and WM_*BUTTONDOWN messages can be L (left mouse button), M (middle mouse button), or R (right mouse button).

An application can change the behavior of a floating pop-up menu displayed in response to a WM_*BUTTONDOWN message to keep it visible after the mouse button is released. However, when an application uses the techniques described below, it changes the menu's user interface. Specifically, to change menu selections with the mouse, the user must first release the mouse button and then press it again. Dragging the mouse between items with the button down, without releasing the button at least once, will not change the selection.

To cause a floating pop-up menu to remain visible after it is displayed in response to a WM_*BUTTONDOWN message, follow these four steps:

1. In the application, allocate a 256 byte buffer to hold the key state.

2. Call the GetKeyboardState function with a far pointer to the buffer

   to retrieve the keyboard state.

3. Set the keyboard state for the VK_*BUTTON index in the keyboard
   state array to 0.

4. Call SetKeyboardState with a far pointer to the buffer to register
   the change with Windows.

The keyboard state array is 256 bytes. Each byte represents the state of a particular virtual key. The value 0 indicates that the key is up, and the value 1 indicates that the key is down. The array is indexed by the VK_ values listed in Appendix A of the "Microsoft Windows Software Development Kit Reference Volume 2" for version 3.0.

The code fragment below changes the state of the VK_LBUTTON to 0 (up) during the processing of a WM_LBUTTONDOWN message. This causes TrackPopupMenu to act as if the menu were displayed as the result of a WM_LBUTTONUP message or of a keystroke. Therefore, the menu remains visible even after the mouse button is released and the WM_LBUTTONUP message is received. Items on this menu can be selected with the mouse or the keyboard.

```
   switch (iMessage)
      {
   case WM_LBUTTONDOWN:
      static BYTE rgbKeyState[256];

      GetKeyboardState(rgbKeyState);
      rgbKeyState[VK_LBUTTON] = 0;      // 0==UP, 1==DOWN
      SetKeyboardState(rgbKeyState);

      // Create the pop-up menu and call TrackPopupMenu.
      break;
      }
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMen

Changing Hypertext Jump Color in Windows Help

PSS ID Number: Q75111

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY
=======

The author of a file to be used with Windows Help can alter the default color of hypertext jump strings. This specification overrides both the default behavior of showing these strings as underlined green text and any color preference listed in the WIN.INI file. This article explains the steps required to make this type of modification.

MORE INFORMATION
================

This feature must be used with care. There are situations when the user may require a jump color other than the default. The profile strings in the WIN.INI file provide an opportunity to specify a color value appropriate for the particular situation. For more information on these settings, query on the following words:

```
prod(winsdk) and jumpcolor
```

To specify the color of a context jump when the RTF Help text is created using Microsoft Word for Windows, follow the six steps listed below. This procedure is a modification of the steps listed on page 17-11 of the "Microsoft Windows Software Development Kit Tools" manual.

1. Place the cursor at the point in the text where the jump term will be entered.

2. Choose Character from the Format menu and specify the double-underline attribute and the desired text color. Word for Windows provides 6 colors, along with auto, black, and white.

3. Type the jump word or words.

4. Choose Character from the Format menu. Turn off the double-underline attribute and choose the default text color and hidden text.

5. Type a percent sign (%), followed by the context string assigned to the topic. For example, JumpText%ContextString, where JumpText is given the desired color and %ContextString is hidden text.

6. Choose Character from the Format menu. Turn off hidden text.

After all topics have been created, save the file as RTF (rich text format) and build the .HLP file. For more information on this process, refer to Chapters 15 through 19 of the "Windows Software Development Kit (SDK) Tools" manual.

It is also possible to modify the six color values that Word for Windows provides as defaults. This can be done by modifying the color table in the RTF header. To do this, load the RTF file into Word for Windows, however, do not convert the file from RTF. For more information on the RTF color table, refer to page 389 of the "Microsoft Word for Windows Technical Reference" (Microsoft Press).

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsHlp

Changing Print Settings Mid-Job

PSS ID Number: Q85679

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

In Windows 3.1 and later, print settings can be changed on a page-by-page basis through the ResetDC() API.

MORE INFORMATION

================

An application can pass a new DEVMODE structure (containing new print settings) to ResetDC() between pages to change the print settings. For example, this function makes it possible to change the paper bin or paper orientation for each page in a print job. Note that ResetDC() cannot be used to change the driver name, device name, or the output port.

Before calling ResetDC(), the application must ensure that all objects (other than stock objects) that were previously selected into the printer device context are selected out.

Additional reference words: 3.10 3.50 4.00 95 dmOrientation dmDefaultSource dmPaperSize hDC WM_DEVMODECHANGE ExtDeviceMode

KBCategory: kbprint

KBSubcategory: GdiPrn

Changing the Controls in a Common Dialog Box

PSS ID Number: Q82299

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

One reason to incorporate the common dialogs library routines into an application is the ability to use the basic functionality of one or more of the common dialogs and tailor it to the needs of a particular application.

All of the predefined controls must be present for the Common Dialogs DLL (COMMDLG.DLL) to properly interact with a dialog box. Each predefined control in the dialog box must retain its control ID value. For these reasons, an application cannot delete unnecessary controls from a dialog box.

To prevent the user from interacting with a given control, move the control off screen by specifying very large coordinate values [for example, (4000, 4000)]. The application must also disable the control to prevent it from receiving the focus when the user uses the TAB key to cycle through the controls. Failing to disable the control can create "mystery" tab stops where the input focus disappears.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Changing the Font Used by Dialog Controls in Windows

PSS ID Number: Q74737

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

In Windows 3.x, there are two ways to specify the font used by dialog controls:

1. The FONT statement can be used in the dialog template to specify the font used by ALL the controls in the dialog box.

   -or-

2. The WM_SETFONT message can be sent to one or more dialog controls during the processing of the WM_INITDIALOG message.

If a font is specified in the dialog template, the controls will use a bold version of that font. The following code demonstrates how to change the font used by dialog box controls to a nonbold font using WM_SETFONT. The font should be deleted with DeleteObject() when the dialog box is closed.

```
HWND hDlg;
HFONT hDlgFont;
LOGFONT lFont;

case WM_INITDIALOG:
    /* Get dialog font and create non-bold version */
    hDlgFont = NULL;
    if ((hDlgFont = (HFONT)SendMessage(hDlg, WM_GETFONT, 0, 0L))
             != NULL)
       {
       if (GetObject(hDlgFont, sizeof(LOGFONT), (LPSTR)&lFont)
                != NULL)
          {
          lFont.lfWeight = FW_NORMAL;
          if ((hDlgFont = CreateFontIndirect(&lFont)) != NULL)
             {
             SendDlgItemMessage(hDlg, CTR1, WM_SETFONT, hDlgFont, 0L);
             // Send WM_SETFONT message to desired controls
             }
          }
       }
    else  // user did not specify a font in the dialog template
       {  // must simulate system font
       lFont.lfHeight = 13;
```

```
        lFont.lfWidth = 0;
        lFont.lfEscapement = 0;
        lFont.lfOrientation = 0;
        lFont.lfWeight = 200; // non-bold font weight
        lFont.lfItalic = 0;
        lFont.lfUnderline = 0;
        lFont.lfStrikeOut = 0;
        lFont.lfCharSet = ANSI_CHARSET;
        lFont.lfOutPrecision = OUT_DEFAULT_PRECIS;
        lFont.lfClipPrecision = CLIP_DEFAULT_PRECIS;
        lFont.lfQuality = DEFAULT_QUALITY;
        lFont.lfPitchAndFamily = VARIABLE_PITCH | FF_SWISS;
        lFont.lfFaceName[0] = NULL;
        hDlgFont = CreateFontIndirect(&lFont);

        SendDlgItemMessage(hDlg, CTR1, WM_SETFONT, hDlgFont,
           (DWORD)TRUE);
        // Send WM_SETFONT message to desired controls
        }

   return TRUE;
   break;
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Changing/Setting the Default Push Button in a Dialog Box

PSS ID Number: Q67655

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The default push button in a dialog box is defined to be the button that is pressed when the user chooses the ENTER key, provided that the input focus is not on another button in the dialog box. The default push button is visually distinguished from other buttons by a thick dark border. This article describes how to change the default push button.

MORE INFORMATION

To change the default push button, perform the following three steps:

1. Send the BM_SETSTYLE message to the current default push button to change its border to that of a regular push button.

2. Send a DM_SETDEFID message to the dialog box to change the ID of the default push button.

3. Send the BM_SETSTYLE message to the new default push button to change its border to that of a default push button.

The following is sample code that performs the three steps:

Sample Code

```
   // Reset the current default push button to a regular button.
   SendDlgItemMessage(hDlg, <ID of current default push button>,
         BM_SETSTYLE, BS_PUSHBUTTON, (LONG)TRUE);

   // Update the default push button's control ID.
   SendMessage(hDlg, DM_SETDEFID, <ID of new default push button>,
         0L);

   // Set the new style.
   SendDlgItemMessage(hDlg, <ID of new default push button>,
         BM_SETSTYLE, BS_DEFPUSHBUTTON, (LONG)TRUE);
```

NOTE: For Win32, the (LONG) casts should be changed to (LPARAM).

Note, however, that ANY push button that has the input focus will have a dark border. A default push button will retain this dark border even when the input focus is transferred to another control in the dialog box, provided the new control is not another push button.

For example, if the input focus is on an edit control, check box, radio button, or any control other than a push button, and the ENTER key is pressed, Windows sends a WM_COMMAND message to the dialog box procedure with the wParam set to the control ID of the default push button.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Character Sets Supported by Hangeul (Korean) Windows Versions

PSS ID Number: Q130055

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 4.0
- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Win32s version 1.2

SUMMARY

Hangeul (Korean) Windows version 3.1 supports the Wansung code set only. However Hangeul Windows 95 will support the XWansung code set.

MORE INFORMATION

Two different Hangeul character standards exist in Korea. One is Wansung, and the other is Johap. The Korean government issued a Korean standard code set (KSC5601-1987), which is Wansung code set. Hangeul Windows version 3.1 supports this character set only.

Later, the Korean government amended and added to the standard and issued a new one (KSC5601-1992), which contains both Wansung and Johap code sets.

Hangeul Windows version 3.1 and Hangeul Windows 95 will not support the KSC5601-1992 standard. Instead, Hangeul Windows 95 will support the XWansung (Microsoft Extended Wansung) standard, which is an extended version of the KSC5601-1987 standard.

The XWansung Code system contains some Johab characters (not all the Johap codes). Johab characters were added into the vacant Range of old Wansung Code of Hangeul Windows version 3.1. Here are the details on the XWansung code set:

Number of characters:

```
     Exisitng Wansung    = 8,836
     Additional Assigned = 8,822 (Johab)
     Additional Reserved = 4,770
     -----------------------------
     Total characters    = 22,428
```

```
Leading byte range:    0x81-0xFE
Trailing byte range:   0x41-5A,0x61-0x7A,0x81-0xFE
```

Additional reference words: 1.20 3.10 4.00 3.50 Hangul Chohap kbinf
KBCategory: kbother
KBSubcategory: wintldev

Choosing the Debugger That the System Will Spawn

PSS ID Number: Q103861

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

SUMMARY
=======

It is possible to have the system spawn a debugger whenever an application faults. The capability is controlled by the following Registry key on Windows NT:

```
HKEY_LOCAL_MACHINE\
          SOFTWARE\
         Microsoft\
        Windows NT\
     CurrentVersion\
           AeDebug
```

This key contains the following entries:

```
Auto
Debugger
```

These entries are also available on Windows 95. However, on Windows 95, they are contained in the win.ini file instead of the registry. The section [aedebug] has entries that correspond to the registry.

If the value of Auto is set to "0" (zero), then the system will generate a pop-up window, and if the user chooses Cancel, spawn the debugger that is specified in the Debugger value. If the value of Auto is set to "1", then the system will automatically spawn the debugger that is specified in the Debugger value.

After installing Windows NT, the Debugger value is set to

```
DRWTSN32 -p %ld -e %ld -g
```

and the Auto value is set to 1. If the Win32 SDK is installed, then the Debugger value is changed to

```
<MSTOOLS>\BIN\WINDBG -p %ld -e %ld
```

and the Auto value is set to 0.

MORE INFORMATION
================

The DRWTSN32 debugger is a post-mortem debugger similar in functionality to the Windows 3.1 Dr. Watson program. DRWTSN32 generates a log file

containing fault information about the offending application. The following data is generated in the DRWTSN32.LOG file:

- Exception information (exception number and name)
- System information (machine name, user name, OS version, and so forth
- Task list
- State dump for each thread (register dump, disassembly, stack walk, symbol table)

A record of each application error is recorded in the application event log. The application error data for each crash is stored in a log file named DRWTSN32.LOG, which by default is placed in your Windows directory.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbtool
KBSubcategory: TlsMisc

Clarification of COMMPROP dwMax?xQueue Members

PSS ID Number: Q94950

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The entry for the COMMPROP structure in the Win32 Programmer's Reference states that for the dwMaxTxQueue and dwMaxRxQueue members, "a value of 0 means that this field is not used".

MORE INFORMATION

This statement means that the provider does not restrict you to maximum Rx and Tx queue values, and these members [returned by GetCommProperties()] should not be used to determine the size of your transmit and receive buffers when calling SetupComm().

Based on the memory present in the system, the Windows NT serial driver determines a default Rx queue size (currently 128 bytes on low memory systems and 4K on high memory systems). The current Rx and Tx queue sizes are located in the dwCurrentTxQueue and dwCurrentRxQueue members.

SetupComm() allows you to change these default queue sizes. However, you should not assume that the given serial driver will allocate any memory. The queue size allocated is stored in the dwCurrentRxQueue member of the COMMPROP structure. You may use this information to set the XonLim and XoffLim members of the device control block (DCB) structure.

The Microsoft-supplied serial driver attempts to allocate at least the amount requested for the RXQUEUE and, failing this, the request will also fail. The driver never attempts to allocate memory for the TXQUEUE.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseCommapi

Clarification of the "Country" Setting

PSS ID Number: Q102765

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5
    - Microsoft Windows 95 version 4.0

Under Windows NT, the "Country" choice affects currency, date/time and number format, and so forth. The "Language" choice affects sorting, names of the days of the weeks and months, and so forth. These settings allow the user to choose the appropriate language and country format. For example, if you are British and living in the U.S., you can pick a locale of English (British) at setup time, then use Control Panel later to change your country to U.S. so that currency is in dollars instead of pounds.

In Windows 95, there is not both a Country and a Language choice. You are asked for a single Regional Setting.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbui
KBSubcategory: UsrNls WintlDev

Clearing a Message Box

PSS ID Number: Q74444

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

During the processing of the MessageBox() function, Windows creates a bitmap to save the part of the screen covered by the message box. Normally, before the MessageBox() function returns, Windows repaints the portion of the screen covered by the message box using the bitmap. In this scenario, when the user clicks on a button to dismiss the message box, the message box disappears immediately.

It is important to note that under low memory conditions, Windows will discard the bitmap. If the bitmap is discarded and a significant amount of processing takes place between the MessageBox() call and painting the application's window, the vestigial image of the message box will remain on the screen during the processing. If the user clicks on this image with the mouse, the underlying window will receive the mouse messages. This can cause unexpected (and possibly undesirable) effects.

To address this problem, call UpdateWindow() immediately after MessageBox(). The parameter to UpdateWindow() should be the parent window of the message box (or of the application's main window if the message box has no parent). This will cause the application to paint the affected window if the bitmap has been discarded. The message box will disappear immediately under all circumstances.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Client Service For Novell Netware Doesn't Support Named Pipes

PSS ID Number: Q129317

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

The Windows NT Client Service for Novell NetWare does not support named pipes. Consequently, 16-bit Novell NetWare client applications running on Windows NT cannot connect to Novell NetWare named pipe server applications. There are currently no plans to add named pipe support to the Windows NT Client Service for Novell NetWare.

For more information on Novell NetWare named pipes, consult the following Novell documentation:

- NetWare Client for OS/2 User's Guide
- NetWare Client for MS-DOS and Microsoft Windows User's Guide

Additional reference words: 3.10 3.50
KBCategory: kbnetwork
KBSubcategory: NtwkMisc

Clipboard Memory Sharing in Windows

PSS ID Number: Q11654

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The following are questions and answers on the topic of Clipboard memory sharing:

Q. Does the Clipboard UNLOCK before freeing the handle when I tell it to SetClipboardData()?

A. Yes, the Clipboard UNLOCKs before freeing the handle when you SetClipboardData().

Q. Does the Clipboard actually copy my global storage to another block, or does it just retain the value of my handle for referencing my block?

A. The Clipboard is sharable; it retains the value of the handle.

Q. Does GetClipboardData() remove the data from the Clipboard, or does it allow me to reference the data without removing it from the Clipboard?

A. The data handle returned by GetClipboardData() is controlled by the Clipboard, not by the application. The application should copy the data immediately, instead of relying on the data handle for long-term use. The application should not free the data handle or leave it locked. To remove data from the Clipboard, call SetClipboardData().

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrClp

Combo Box w/Edit Control & Owner-Draw Style Incompatible

PSS ID Number: Q82078

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The owner-draw combo box styles (CBS_OWNERDRAWFIXED and CBS_OWNERDRAWVARIABLE) are incompatible with the combo box styles that contain an edit control (combo box styles CBS_SIMPLE and CBS_DROPDOWN). A combo box with either the CBS_SIMPLE or CBS_DROPDOWN style displays the currently selected item in its associated edit control. When an owner-draw style is specified for the combo box style CBS_SIMPLE or CBS_DROPDOWN, the current selection may not be displayed. Using the SetWindowText function to display the current selection in response to a CBN_SELCHANGE message may not be effective.

MORE INFORMATION

An owner-draw combo box can contain bitmaps or other graphic elements in its list box. Therefore, to correctly display the current selection, it is necessary to display a bitmap or other graphic element in the edit control. Because edit controls are not designed to display graphics, there is no natural method to display the current selection in an owner-draw combo box with an edit control.

The combo box style CBS_DROPDOWNLIST, which has a static text area instead of an edit control, can display any item, including graphics. Use this style combo box with the owner-draw styles.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

COMCTL32 APIs Unsupported in the Win32 SDK

PSS ID Number: Q105300

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Win32s version 1.2

The library COMCTL32.LIB was included in the Win32 SDK because the PERFMON sample makes use of it. The library is part of the Windows for Workgroups (WFW) COMMCTRL.LIB. However, Microsoft does not officially support COMCTL32.LIB or recommend the use of these application programming interfaces (APIs), and therefore they have not been documented in the SDK.

Microsoft does not recommend using these APIs, because Microsoft is providing the new controls in Windows 95, Windows NT 3.51, and Win32s 1.3.

If you must absolutely use COMCTL32 with earlier versions, the documentation can be found in the WFW SDK. Be aware that these APIs are unsupported, and code that you write will not work on Windows 95 and Windows NT 3.51.

Additional reference words: 1.20 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrCtl W32s

Common Dialog Boxes and the WM_INITDIALOG Message

PSS ID Number: Q74610

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

SUMMARY

An application using the common dialog box library (COMMDLG.DLL) can override any information initialized in the DLL by handling the WM_INITDIALOG message in its dialog hook function. If the application is using a private dialog template, it should also initialize all private dialog items while handling this message.

After processing the WM_INITDIALOG message, the hook function should return FALSE if it has set the focus to a dialog control, and return TRUE if Windows should set the focus.

MORE INFORMATION

For example, consider an application that is using the Open File common dialog box (via GetOpenFileName()) but does not want the Drives combo box to appear in the dialog box. Since all dialog items in the standard dialog template must be included in the application's private dialog template, the application will need to include code to disable and hide the Drives combo box and the corresponding "Drives:" static text control. This code would be implemented in the WM_INITDIALOG case of the dialog hook function, as follows:

```
   case WM_INITDIALOG:
      hWnd = GetDlgItem( hDlg, cmb2 );  // Get Drives combo box handle
      EnableWindow( hWnd, FALSE );      // No longer receives input,
                                        // no longer a tabstop
      SetWindowPos( hWnd, NULL, 0, 0, 0, 0, SWP_HIDEWINDOW );

      hWnd = GetDlgItem( hDlg, stc4 );  // Get "Drives:" static control
      EnableWindow( hWnd, FALSE );      // no longer an accelerator
      SetWindowPos( hWnd, NULL, 0, 0, 0, 0, SWP_HIDEWINDOW);

   // Initialize private dialog items here...

      return( TRUE );                   // Let Windows set the focus
```

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Common File Mapping Problems and Platform Differences

PSS ID Number: Q125713

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY
=======

This article addresses some common problems encountered when using file mapping. It also points out some platform differences in the file mapping implementation.

This article does not describe the procedures for performing file mapping. For information on using file mapping, please see the File Mapping overview in the Microsoft Win32 Programmer's Reference. Also see the descriptions for CreateFileMapping(), OpenFileMapping(), MapViewOfFile(), MapViewOfFileEx(), UnmapViewOfFile(), and FlushViewOfFile().

MORE INFORMATION
================

Name Space Conflicts

The names of event, semaphore, mutex, and file-mapping objects share the same name space, so it is not possible to have two different object types with the same name. It is an error to attempt to create or open an object of one type using a name that is already being used by an object of another type.

CreateFileMapping() and OpenFileMapping() will fail if they specify an object name that is in use by an object of another type. In both cases, GetLastError() will return ERROR_INVALID_HANDLE (6).

To avoid conflicts between object types, one solution is to include the object type in the name. For example, use "EV_myapp_block_ready" for an event object name and "FM_myapp_missile_data" for a file mapping object name.

Necessity of Unmapping All Views of a Mapped File

Windows maintains an internal handle to a file mapping object for each view of that object, whether created by MapViewOfFile() or MapViewOfFileEx(). This internal handle is kept in addition to the handle returned by CreateFileMapping(). The internal handle is not closed until the view associated with the handle is unmapped by calling

UnmapViewOfFile(). To completely close a file mapping object requires that all handles for the object, including internal handles, be closed. Thus, to close a file mapping object, all views of that object must be unmapped, and the handle returned by CreateFileMapping() must be closed.

Extant unmapped views of a file mapping object will NOT cause a CloseHandle() on the object's handle to fail. In other words, when your handle to the object is closed successfully, it is not necessarily true that all views have been unmapped, so the file mapping object has not necessarily been freed.

Failure to properly unmap all views of the object and to close the handle to the object will cause leaks in the application's paged pool, nonpaged pool, virtual bytes, and also in the system wide committed bytes.

Restrictions on the Size of File Mapping Objects

The size of a file mapping object backed by the system paging file is limited to available system virtual memory (meaning the amount of memory that could be committed with a call to VirtualAlloc()).

On Windows NT, the size of a file mapping object backed by a named disk file is limited by available disk space. The size of a mapped view of an object is limited to the largest contiguous block of unreserved virtual memory in the process performing the mapping (at most, 2GB minus the virtual memory already reserved by the process).

On Win32s, the size of a file mapping object backed by a named disk file is limited to available system virtual memory, due to the virtual memory management implementation of Win32s. Win32s allocates regular virtual memory for the memory mapped section even though it does not need swap space, and the amount of VM set by Windows is too small to use for mapping large files. As with Windows NT, available disk space will also impose a limitation.

On Windows 95, the size of a file mapping object backed by a named disk file is limited to available disk space. The size of a mapped view of an object is limited to the largest contiguous block of unreserved virtual memory in the shared virtual arena. This block will be at most 1GB, minus any memory in use by other components of Windows 95 which use the shared virtual arena (such as 16-bit Windows-based applications). Each mapped view will use memory from this arena, so this limit applies to the total size of all non-overlapping mapped views for all applications running on the system.

Mapped File May Not be Automatically Grown

If the size specified for a file mapping object backed by a named disk file in a call to CreateFileMapping() is larger than the size of the file used to back the mapping, the file will normally be grown to the specified size by the CreateFileMapping() call.

On Windows NT only, if PAGE_WRITECOPY is specified for the fdwProtect

parameter, the file will not automatically be grown. This will cause CreateFileMapping() to fail, and GetLastError() will return ERROR_NOT_ENOUGH_MEMORY (8). To set the size of the file before calling CreateFileMapping(), use SetFilePointer() and SetEndOfFile().

MapViewOfFileEx() and Valid Range of lpvBase

On Windows NT, views of file mapping objects are mapped in the address range of 0-2 GB. Passing an address outside of this range as the lpvBase parameter to MapViewOfFileEx() will cause it to fail, and GetLastError() will return ERROR_INVALID_PARAMETER (87).

On Windows 95, views of file mapping objects are mapped in the address range of 2-3 GB (the shared virtual arena). Passing an address outside of this range will cause MapViewOfFileEx() to fail, and GetLastError() will return ERROR_INVALID_ADDRESS (487). Note that future updates to Windows 95 may change the mapping range to 0-2 GB, as on Windows NT.

MapViewOfFileEx() and Allocation Status of lpvBase

If an address is specified for the lpvBase parameter of MapViewOfFileEx(), and there is not a block of unreserved virtual address space at that address large enough to satisfy the number of bytes specified in the cbMap parameter, then MapViewOfFileEx() will fail, and GetLastError() will return ERROR_NOT_ENOUGH_MEMORY (8). This does not mean that the system is low on memory or that the process cannot allocate more memory. It simply means that the virtual address range requested has already been reserved in that process.

Prior to calling MapViewOfFileEx(), VirtualQuery() can be used to determine an appropriate range of unreserved virtual address space.

MapViewOfFileEx() and Granularity of lpvBase

For the lpvBase parameter specified in a call to MapViewOfFileEx(), you should use an integral multiple of the system's allocation granularity. On Windows NT, not specifying such a value will cause MapViewOfFileEx() to fail, and GetLastError() to return ERROR_MAPPED_ALIGNMENT (1132). On Windows 95, the address is rounded down to the nearest integral multiple of the system's allocation granularity.

To determine the system's allocation granularity, call GetSystemInfo().

Addresses of Mapped Views

When mapping a view of a file (or shared memory), it is possible to either let the operating system determine the address of the view, or to specify an address as the lpvBase parameter of the MapViewOfFileEx() function. If the file mapping is going to be shared among multiple processes, then the recommended method is to use MapViewOfFile() and let the operating system select the mapping address for you. There are good reasons for doing

so:

- On Windows NT, views are mapped independently into each process's address space. While it may be convenient to try to map the view at the same address in each process, the specified virtual address range may not be free in all of the processes involved. Therefore, the mapping could fail in one (or more) of the processes trying to share the file mapping.

- On Windows 95, file mapping objects exist in the 2-3 GB address range (the shared virtual arena). Therefore, once the initial address for the view is determined, additional views of the mapping will be mapped to the same address in each process anyway, and there is no benefit in trying to force the initial mapping to a specific address. For the second and subsequent views of a mapping object, if the address specified for lpvBase does not match the actual address where Windows 95 has mapped the view, then MapViewOfFileEx() fails, and GetLastError() returns ERROR_INVALID_ADDRESS (487). Additionally, when attempting to map the first view at a pre-determined address, that address may already be in use by other components of Windows 95 which use the shared virtual arena. Note that future updates to Windows 95 may change the mapping range to 0-2 GB, as on Windows NT.

If it is absolutely necessary to create the mappings at the same address in multiple processes under Windows NT, here are two possible approaches:

1. Pick an appropriate address and manage the virtual address space so that this address is left available. This means basing your DLLs, allocating memory at specific locations, and using a tool such as Process Walker to observe the virtual address space pattern. As soon as possible in the execution of the application, either reserve the desired address space or perform the mapping. One good place to do this is in the PROCESS_ATTACH handling in a DLL, because it is called before the executable itself is started. NOTE: There is still no guarantee that some DLL will not have already loaded at the address in question. If not all involved processes can map at the predetermined address, they can either fail or try a new address.

   -or-

2. Have all processes involved negotiate an appropriate address. The processes can all use the VirtualQuery() function to scan their address spaces until a common address is found in each process that has a large enough unreserved block. This requires that all processes involved map the address at the same time. A process that starts after the address has been determined must map at that address, and fail if it cannot do so. Alternatively, the negotiation process could be repeated, with each process remapping at the new address. Then, all pointers into the mapping must be readjusted.

The second method is far more likely to succeed. It can also be combined with the first to make it more likely that an appropriate address will be found quickly.

When views are mapped to different addresses under Windows NT, the

difficulty that arises is storing pointers to the mapping within the mapping itself. This is because a pointer in one process does not point to the same location within the mapping in another process. To overcome this problem, store offsets rather than pointers in the mapping, and calculate actual addresses in each process by adding the base address of the mapping to the offset. It is also possible to used based pointers and thus perform the base + offset conversion implicitly. A short SDK sample called BPOINTER demonstrates this technique.

Additional Platform Differences

Additional limitations when performing file mapping under Windows 95:

1. The dwOffsetHigh parameters of MapViewOfFile() and MapViewOfFileEx() are not used, and should be zero. Windows 95 uses a 32-bit file system.

2. The dwMaximumSizeHigh parameter of CreateFileMapping() is not used, and should be zero. Again, this is due to the 32-bit file system.

3. The SEC_IMAGE and SEC_NOCACHE flags for the fdwProtect parameter of CreateFileMapping() are not supported.

4. If the FILE_MAP_COPY flag is used to map a view of a file mapping object, the object must have been created using PAGE_WRITECOPY protection. Additionally, the object must be backed by a named file rather than the system paging file (in other words, a valid file handle, not (HANDLE)0xFFFFFFFF, must be specified for the hFile parameter of CreateFileMapping()). Failure to do either of these causes MapViewOfFile() to fail, and GetLastError() to return ERROR_INVALID_PARAMETER (87).

5. If two or more processes map a PAGE_WRITECOPY view of the same file mapping object (by using a named object, for example), they are able to see changes made to the view by the other process(es). The actual disk file is not modified, however. Under Windows NT, if one process writes to the view, it receives its own copy of the modified pages and will not affect the pages in the other process(es) or the disk file.

Additional reference words: 1.30 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMm

Complete Enumeration of System Fonts

PSS ID Number: Q99672

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Complete enumeration of system fonts is a two-phase process. Applications should first call EnumFontfamilies with NULL as the family name to enumerate all the font face names in the system. Applications should then take each face name and call EnumFontFamilies again to obtain the style names (for TrueType fonts only) or the supported point sizes (for raster fonts only). The style names are not supported for the raster and vector fonts. Because TrueType and vector fonts are continuously scalable, their point sizes are not enumerated.

MORE INFORMATION

================

The following steps detail the enumeration:

1. Call EnumFontFamilies with NULL as the family name (lpszFamily) to list one font from each available font family.

2. In the EnumFontFamProc callback function, look at the nFontType parameter.

3. If nFontType has the TRUETYPE_FONTTYPE flag set, then call EnumFontFamilies with the family name set to the font's type face name (lfFaceName of the ENUMLOGFONT structure). The callback function is called once for each style name. This enumeration is useful if the application is interested in finding a TrueType font with a specific style name (such as "Outline"). Because a TrueType font is continuously scalable, it is not necessary to enumerate a given font for point sizes. An application may use any desired point size. If the application is listing the enumerated TrueType fonts, it can simply choose some representative point sizes in a given range. The point sizes recommended by "The Windows Interface: An Application Design Guide" (page 159, Section 8.4.1.4) are 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26, 28, 36, 48, and 72. All TrueType fonts are available on both displays and printers, so an application can be sure that the font appears the same on the display and the printer.

4. If nFontType has the RASTER_FONTTYPE flag set, then call EnumFontFamilies with the family name set to the font's type face name. The callback function is called once for each available point size. Raster fonts can be scaled only in multiples of the available point sizes. Because scaled raster fonts are usually not appealing to the user, applications may choose to limit themselves to the available sizes. Because Microsoft Windows version 3.1 does not define style names for raster fonts, there is no need to enumerate for style names.

   If the nFontType also has the DEVICE_FONTTYPE flag set, then the current font is a raster font available to the printer driver for use with the printer. The printer may have these fonts in hardware or be capable of downloading them when necessary. Applications that use such fonts should be aware that similar raster fonts may not be available on the display device. The converse is also true. If the DEVICE_FONTTYPE flag is not set, then applications should be aware that a similar font may not be available on the printer. Fonts generated by font packages such as Adobe Type Manager (ATM) are listed as device fonts.

5. If nFontType has neither the TRUETYPE_FONTTYPE nor the RASTER_FONTTYPE flags set, then the enumerated font is a vector font. Vector fonts are also continuously scalable so they do not have to be enumerated for point sizes. Because Windows 3.1 does not support style names for vector fonts, there is no need to enumerate them for style names. Vector fonts are generally used by devices such as plotters that cannot support raster fonts. These fonts generally have a poor appearance on raster devices, so many applications avoid them.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Computing the Size of a New ACL

PSS ID Number: Q102103

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

When adding an access-allowed access control entry (ACE) to a discretionary access control list (DACL), it is useful to know the exact size needed for the new DACL. This is particularly useful when creating a new DACL and copying over the existing ACEs. The below code computes the size needed for a DACL if an access-allowed ACE is added:

```
   ACL_SIZE_INFORMATION AclInfo;

   GetAclInformation(pACL,&AclInfo,sizeof(ACL_SIZE_INFORMATION),
                     AclSizeInformation))

   dwNewACLSize = AclInfo.AclBytesInUse +
                  sizeof(ACCESS_ALLOWED_ACE) +
                  GetLengthSid(UserSID) - sizeof(DWORD);
```

MORE INFORMATION
================

The call to GetAclInformation() takes a pointer to an ACL. This point is supplied by your program and should point to the DACL you want to add an access-allowed ACE to. The GetAclInformation() call fills out a ACL_SIZE_INFORMATION structure, which provides size information on the ACL.

The second statement computes what the new size of the ACL will be if an access-allowed ACE is added. This is accomplished by adding the current bytes being used to the size of an ACCESS_ALLOWED_ACE. We then add the size of the security identifier (SID) (provided by your application) that is to used in the AddAccessAllowedAce() API call. Subtracting out the size of a DWORD is the final adjustment needed to obtain the exact size. This adjust is to compensate for a place holder member in the ACCESS_ALLOWED_ACE structure which is used in variable length ACEs.

When adding an ACE to an existing ACL, often there is not enough free space in the ACL to accommodate the additional ACE. In this situation, it is necessary to allocate a new ACL and copy over the existing ACEs and then add the access-allowed ACE. The above code can be used to determine the amount of memory to allocate for the new ACL.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Conditionally Activating a Button in Windows Help

PSS ID Number: Q76534

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY
=======

An application can add buttons to the Windows Help engine button bar. There may be times when the button should be activated or deactivated based on an external condition.

For example, if an application adds a tutorial button to the bar and the user has not chosen to install the tutorial, the button should be dimmed to indicate that the tutorial is not available.

MORE INFORMATION
================

The following code fragment demonstrates activating and deactivating buttons using the macro facility:

```
char szMacro[255];
    .
    .
    .
/* Bring up the Help engine with the HLP file   */
/* This code fragment assumes that a button has */
/* been defined with an ID of TUTORIAL_BUTTON   */
WinHelp (hWnd, lpHelpFile, HELP_CONTENTS, 0L)

if (fTutorial)
/* If the tutorial is installed, the macro should enable the button */
     lstrcpy(szMacro,"EnableButton(`TUTORIAL_BUTTON')");
else
/* If the tutorial is not installed, the macro should disable the
   button */
    lstrcpy(szMacro,"DisableButton(`TUTORIAL_BUTTON')");

/* Note that the single quote character before TUTORIAL must be a backquote
/* (`) for the function to work correctly.

/* Run the appropriate macro */
WinHelp (hWnd, lpHelpFile, HELP_COMMAND, (LONG)szMacro);
```

Additional reference words: 3.10 3.50 4.00 95 grayed out disabled unavailable
KBCategory: kbtool
KBSubcategory: TlsHlp

Configuration Needed to Run RPC Apps Using IPX Protocol

PSS ID Number: Q139559

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, and 4.0

SUMMARY

This article describes certain conmponents that need to be installed when running RPC applications using the IPX/SPX protocol.

MORE INFORMATION

All the information below assumes that the IPX/SPX protocol is installed and the 16-bit runtimes are installed where needed.

Platform	Application	Protocol Sequence	Component needed
Windows NT	Server	ncadg_ipx	Client Services for Netware(WKS) or Gateway Services for Netware(SVR)
Windows NT	Server	ncacn_spx	Client Services for Netware(WKS) or Gateway Services for Netware(SVR)
Windows 95	Server	ncacn_spx	no additional component
Windows NT	Client(32-bit)	ncadg_ipx	SAP agent; else use IPX addresses (see note (2) below) else Server must have Gateway Services for Netware
Windows NT	Client(32-bit)	ncacn_spx	as above
Windows 95	Client(32-bit)	ncacn_spx	Client for Novell Netware else use IPX addresses (see note (2) below)
Windows NT	Client(16-bit)	ncadg_ipx	SAP agent; else use IPX addresses (see note (2) below) else Server

			must have Gateway Services for Netware
Windows NT	Client(16-bit)	ncacn_spx	as above
Windows 95	Client(16-bit)	ncacn_spx	Only way to connect to server is by using IPX addresses (see note (2) below)

NOTES:

1. Windows 95 does not support ncadg_ipx.

2. IPX addresses : The IPX address of the computer must be of the form ~XXXXXXXXYYYYYYYYYYYY. For example, if a computer's network address is 00006112 and its node address is 00AA0060DA28, you should use ~0000611200AA0060DA28.

Additional reference words: 3.10 3.50 4.00 win16sdk Windows 95
KBCategory: kbnetwork kbtshoot
KBSubcategory: NtwkRpc

Consequences of Using CreateDIBSection() With DIB_PAL_COLORS

PSS ID Number: Q137371

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.5, 3.51, 4.0

When using CreateDIBSection() with DIB_PAL_COLORS, the GDI will use the color indices in the bmiColors to populate the color table of the DIB section by resolving the indices to the RGB values from the palette selected into the device context passed into CreateDIBSection(). Consequently, after a program calls CreateDIBSection(), the color table will contain RGB values rather than palette indices. Specifying DIB_PAL_COLORS affects only how the color table is initialized, not how it is subsequently used.

Additional reference words: 4.00 3.50 hbitmap
KBCategory: kbgraphic
KBSubcategory: W32

Considerations for CreateCursor() and CreateIcon()

PSS ID Number: Q73667

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

An application can use the CreateCursor() and CreateIcon() APIs to create icons and cursors on the fly. The application determines the shape at run time.

When the shape of the icons and the cursors is known in advance, an application should use LoadIcon() and LoadCursor().

An application that uses CreateIcon() must call DestroyIcon() to free the memory used by the icon when it is no longer needed. An application that uses CreateCursor() must call DestroyCursor() to release the memory used by the cursor when it is no longer needed.

An application can call DestroyIcon() and DestroyCursor() only when the icon or the cursor is not in use. For example, if the cursor created by CreateCursor() has been specified in a SetCursor call, it must not be destroyed until it has been released by another SetCursor() call.

An application can only use DestroyIcon() and DestroyCursor() to destroy icons and cursors created by CreateIcon() and CreateCursor(). It should not try to destroy icons and cursors loaded with LoadIcon() and LoadCursor().

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiCurico

Console App Launched from a Service Exits When User Logs Off

PSS ID Number: Q149901

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), for Windows NT, versions 3.5, 3.51, 4.0

SUMMARY

=======

Console applications contain a default console control handler. The console control handler function is called by the system when a process receives a CTRL_C, CTRL_BREAK, CTRL_CLOSE, CTRL_LOGOFF, or CTRL_SHUTDOWN signal. By default, when the console control handler receives any one of these signals, it calls ExitProcess().

When a console application is launched from a service, the application receives a modified default console control handler. The modified handler does not call ExitProcess() for the CTRL_LOGOFF signal. This allows a console application to continue running when a user logs off the system. This corresponds with Windows NT services that run when no user is logged onto the system.

If the console application being launched from a service installs its own console control handler, this handler is called before the default handler. If the installed handler calls ExitProcess() as a result of a CTRL_LOGOFF signal, the console application exits when the user logs off the system.

Using any third party libraries or DLLs could cause a console control handler to be installed for an application. If installed, this handler overrides the default handler, causing the application to exit when receiving a CTRL_LOGOFF signal.

For example, if you install a SIGBREAK handler via the C run-time function signal(), the function installs a handler which overrides the default handler. This causes the application to exit when receiving a CTRL_LOGOFF event even though it was launched from a service. This problem can be resolved by either not installing the SIGBREAK handler or by installing another console control handler after the call to signal(). The new console control handler must intercept the CTRL_LOGOFF event and just return TRUE for the application not to exit.

REFERENCES

==========

For more information on console control handlers, please see the Console Overview in the Win32 SDK documentation.

Additional reference words: 3.50 3.51 4.00

KBCategory: kbprg

KBSubcategory: BseService BseCon

Consoles Do Not Support ANSI Escape Sequences

PSS ID Number: Q84240

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

Windows NT does not support ANSI escape sequences. There is some functionality that this affects (for example, changing the color of the prompt). This also affects a very limited number TTY-type programs that rely on the console for escape support to be provided.

This feature is is under review and is being considered for future releases.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseCon

Controlling the Caret Color

PSS ID Number: Q84054

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

When an application creates a custom caret using a bitmap, it is possible to specify white or black for the caret color. In the case of Windows running on a monochrome display, the application can cause the caret to be the color of the display (white, green, amber, and so forth, as appropriate). However, because the caret color is determined by Windows at run time based on the hardware installed, the application cannot guarantee what color will be used under all circumstances. This article provides information about using color in a custom caret.

MORE INFORMATION
================

To create a caret, first create a bitmap with the desired pattern. To display the caret, Windows exclusive-ORs (XORs) and takes the opposite of the result (the NOT of the result) of the bitmap with the background of the client window. Therefore, to create a white caret, create a bitmap that when XOR'd with the window background will have an opposite value that will create a white color. It is when the caret blinks that Windows uses the reverse of the bitmap XOR'd with the background to draw the caret; this creates the white blink seen on the screen.

The bitmap for the caret cannot use a color palette. Windows does not use the color values from the palette in its calculations but the indices into the palette. While it is possible to use palette indices successfully, perfect symmetry of the colors in the palette is required. This is unlikely. For each color in the palette, its exact opposite color must be in the palette, in the exactly opposite index position.

However, when the application creates bitmaps itself, it has complete control over the bits. Therefore, the application can create the perfect counterpart that corresponds to the window background color. If the application uses this information to create the caret bitmap, when Windows creates the caret, it can choose the closest color available in the system palette.

Therefore, to create a white caret (or a black one, if the screen has too many light elements), the task is straightforward. Windows always reserves a few colors in the system palette and makes them available to all applications. On a color display, these colors include black and white. On a monochrome display, these colors are whatever the monochrome color elements are.

Because black and white (or the monochrome screen colors) are always available, the application simply creates a bitmap that, when XOR'd with the screen background color, produces black or white.
The technique involves one main principle: background XOR background = FALSE. Anything XOR'd with itself returns FALSE, which in bitmap terms maps to the color black.

The process of creating a caret from the background color involves the four steps discussed below:

1. Create a pattern brush the same as the window background

2. Select the pattern brush into a memory display context (DC)

3. Use the PATCOPY option of the PatBlt function to copy the brush pattern into the caret bitmap.

4. Specify the caret bitmap in a call to the CreateCaret function.

When this caret is XOR'd with the background, black will result. When the caret blinks, and is therefore displayed, Windows computes the opposite of the caret and XOR's this value into the background. This yields NOT(background XOR background) = NOT(FALSE) = TRUE which corresponds to WHITE. The first background represents the caret bitmap and the second is the current background color of the window.

Note that half the time the custom bitmap is displayed (when the caret "blinks") the other half of the time the background is displayed, (between "blinks").

If the background color is light gray or lighter [RGB values (128, 128, 128) through (255, 255, 255)], then a black caret is usually desired. The process of creating a black caret is just as straightforward. Modify step 1 of the process given above to substitute the inverse of the background for the background bitmap. When the caret blinks, it will show black. The equation that corresponds to this case is NOT(inverse of background XOR background) = NOT(TRUE) = FALSE which corresponds to BLACK.

To change the caret color to something other than black or white requires considerably more work, with much less reliable results because the application must solve the following equation:

```
   NOT(caret XOR background) = desired_color on the
                               "blink" of the caret.
```

where the value for the caret color must be determined given the desired color. A series of raster operations is required to solve this

type of equation. (For more information on raster operations, see chapter 11 of the "Microsoft Windows [3.0] Software Development Kit Reference, Volume 2" or pages 573-585 of the "Microsoft Windows [3.1] Software Development Kit Programmer's Reference, Volume 3: Messages, Structures, and Macros.")

Even after solving this equation, the color actually displayed is controlled by Windows and the colors in the current system palette. With colors other than black or white, an exact match for the desired color may not be available. In that case, Windows will provide the closest match possible. Because the palette is a dynamic entity and can be modified at will, it is impossible to guarantee a particular result color in all cases. The colors black and white should be safe most of the time because it is quite unusual for an application to modify the reserved system colors. Even when an application does change the system palette, it most likely retains a true black and a true white.

As long as a black and white remain in the palette (which is usually the case), this algorithm will provide a white or black caret.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCrt

Converting a Linear Address to a Flat Offset on Win32s

PSS ID Number: Q115080

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, and 1.2

SUMMARY
=======

Win32s flat code and data selectors are not zero-based. Linear addresses retrieved through a VxD can be used in a Win32-based application running under Win32s, after one small change is made.

In addition, there are two Universal Thunk APIs that are used to convert segmented addresses to flat addresses and vice versa.

MORE INFORMATION
================

Linear Address to Flat Address

Win32s does not base linear addresses at 0, so that exceptions will be generated when null pointers are dereferenced. Therefore, an access violation occurs when:

1. a 16-bit DLL calls a VxD to retrieve a linear address (the VxD got the address by translating a physical address to a linear address) through DPMI function 0800h (map physical to linear).

2. the 16-bit DLL returns the address to a Win32-based application through the Universal Thunk.

3. the Win32-based application uses this linear address.

In order to convert a linear address (based at 0) to a flat offset, add the base to the linear address. To do this, get the offset through GetThreadSelectorEntry() with the DS or CS and then subtract that base from the linear address that was returned by the VxD.

Segmented Address to Flat Address

The following Win32s Universal Thunk APIs are used for address translation:

- UTSelectorOffsetToLinear (segmented address to flat address)

- UTLinearToSelectorOffset (flat address to segmented address)

NOTE: In the nested function call

```
   UTLinearToSelectorOffset( UTSelectorOffsetToLinear( x ) );
```

where x is a segmented address, you may not necessarily get the original value of x back. It is by design that the sel:off pair may be different. If the memory was allocated by a 16-bit application, Win32s does not have x in its LinearAddress->selector translation tables. Therefore, when UTLinearToSelectorOffset() is called, new selectors are created.

Additional reference words: 1.10 1.20 gpf gp-fault
KBCategory: kbprg
KBSubcategory: W32s

Converting Colors Between RGB and HLS (HBS)

PSS ID Number: Q29240

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The code fragment below converts colors between RGB (Red, Green, Blue) and HLS/HBS (Hue, Lightness, Saturation/Hue, Brightness, Saturation).

MORE INFORMATION
================

/* Color Conversion Routines --

RGBtoHLS() takes a DWORD RGB value, translates it to HLS, and stores the results in the global vars H, L, and S. HLStoRGB takes the current values of H, L, and S and returns the equivalent value in an RGB DWORD. The vars H, L, and S are only written to by:

1. RGBtoHLS (initialization)
2. The scroll bar handlers

A point of reference for the algorithms is Foley and Van Dam, "Fundamentals of Interactive Computer Graphics," Pages 618-19. Their algorithm is in floating point. CHART implements a less general (hardwired ranges) integral algorithm.

There are potential round-off errors throughout this sample. ((0.5 + x)/y) without floating point is phrased ((x + (y/2))/y), yielding a very small round-off error. This makes many of the following divisions look strange.
*/

```
#define  HLSMAX   RANGE /* H,L, and S vary over 0-HLSMAX */
#define  RGBMAX   255   /* R,G, and B vary over 0-RGBMAX */
                        /* HLSMAX BEST IF DIVISIBLE BY 6 */
                        /* RGBMAX, HLSMAX must each fit in a byte. */

/* Hue is undefined if Saturation is 0 (grey-scale) */
/* This value determines where the Hue scrollbar is */
/* initially set for achromatic colors */
#define UNDEFINED (HLSMAX*2/3)
```

```
void  RGBtoHLS(lRGBColor)
DWORD lRGBColor;
{
   WORD R,G,B;          /* input RGB values */
   BYTE cMax,cMin;      /* max and min RGB values */
   WORD  Rdelta,Gdelta,Bdelta; /* intermediate value: % of spread from max
*/

   /* get R, G, and B out of DWORD */
   R = GetRValue(lRGBColor);
   G = GetGValue(lRGBColor);
   B = GetBValue(lRGBColor);

   /* calculate lightness */
   cMax = max( max(R,G), B);
   cMin = min( min(R,G), B);
   L = ( ((cMax+cMin)*HLSMAX) + RGBMAX )/(2*RGBMAX);

   if (cMax == cMin) {           /* r=g=b --> achromatic case */
      S = 0;                     /* saturation */
      H = UNDEFINED;             /* hue */
   }
   else {                        /* chromatic case */
      /* saturation */
      if (L <= (HLSMAX/2))
         S = ( ((cMax-cMin)*HLSMAX) + ((cMax+cMin)/2) ) / (cMax+cMin);
      else
         S = ( ((cMax-cMin)*HLSMAX) + ((2*RGBMAX-cMax-cMin)/2) )
            / (2*RGBMAX-cMax-cMin);

      /* hue */
      Rdelta = ( ((cMax-R)*(HLSMAX/6)) + ((cMax-cMin)/2) ) / (cMax-cMin);
      Gdelta = ( ((cMax-G)*(HLSMAX/6)) + ((cMax-cMin)/2) ) / (cMax-cMin);
      Bdelta = ( ((cMax-B)*(HLSMAX/6)) + ((cMax-cMin)/2) ) / (cMax-cMin);

      if (R == cMax)
         H = Bdelta - Gdelta;
      else if (G == cMax)
         H = (HLSMAX/3) + Rdelta - Bdelta;
      else /* B == cMax */
         H = ((2*HLSMAX)/3) + Gdelta - Rdelta;

      if (H < 0)
         H += HLSMAX;
      if (H > HLSMAX)
         H -= HLSMAX;
   }
}

/* utility routine for HLStoRGB */
WORD HueToRGB(n1,n2,hue)
WORD n1;
WORD n2;
WORD hue;
```

```
{

   /* range check: note values passed add/subtract thirds of range */
   if (hue < 0)
      hue += HLSMAX;

   if (hue > HLSMAX)
      hue -= HLSMAX;

   /* return r,g, or b value from this tridrant */
   if (hue < (HLSMAX/6))
      return ( n1 + (((n2-n1)*hue+(HLSMAX/12))/(HLSMAX/6)) );
   if (hue < (HLSMAX/2))
      return ( n2 );
   if (hue < ((HLSMAX*2)/3))
      return ( n1 + (((n2-n1)*(((HLSMAX*2)/3)-hue)+(HLSMAX/12))/(HLSMAX/6))
);
   else
      return ( n1 );
}


DWORD HLStoRGB(hue,lum,sat)
WORD hue;
WORD lum;
WORD sat;
{
   WORD R,G,B;                /* RGB component values */
   WORD  Magic1,Magic2;       /* calculated magic numbers (really!) */

   if (sat == 0) {            /* achromatic case */
      R=G=B=(lum*RGBMAX)/HLSMAX;
      if (hue != UNDEFINED) {
         /* ERROR */
      }
   }
   else  {                    /* chromatic case */
      /* set up magic numbers */
      if (lum <= (HLSMAX/2))
         Magic2 = (lum*(HLSMAX + sat) + (HLSMAX/2))/HLSMAX;
      else
         Magic2 = lum + sat - ((lum*sat) + (HLSMAX/2))/HLSMAX;
      Magic1 = 2*lum-Magic2;

      /* get RGB, change units from HLSMAX to RGBMAX */
      R = (HueToRGB(Magic1,Magic2,hue+(HLSMAX/3))*RGBMAX +
(HLSMAX/2))/HLSMAX;
      G = (HueToRGB(Magic1,Magic2,hue)*RGBMAX + (HLSMAX/2)) / HLSMAX;
      B = (HueToRGB(Magic1,Magic2,hue-(HLSMAX/3))*RGBMAX +
(HLSMAX/2))/HLSMAX;
   }

   return(RGB(R,G,B));
}
```

Additional reference words: 3.00 3.10 3.50 4.00 95 color RGB HLS HBS
KBCategory: kbgraphic
KBSubcategory: GdiPal

Copy on Write Page Protection for Windows NT

PSS ID Number: Q103858

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

The Windows NT's Copy on Write page protection is a concept that allows multiple applications to map their virtual address spaces to share the same physical pages, until an application needs to modify the page and have its own instance copy. This is part of a technique called Lazy Evaluation, which allows the system to not waste time by committing resources, time, or execution until/unless absolutely necessary. Copy on Write allows the virtual memory manager to save memory and execution time.

MORE INFORMATION

Copy on Write works as follows: In generic terms, an application can load something into its virtual memory (for example, a code section or DLL code). Virtual memory is mapped to physical memory. Another process may want to load the same thing into its virtual memory. As long as neither process writes to this memory, they can map to, and share, the same physical pages.

If either process needs to write to this memory, because the memory is marked as Copy on Write, the physical page frame will be copied somewhere else in physical memory. Fixups are made for the virtual memory mapping of the writing process. Both applications now have their own instance of the memory contents. In short, applications can share the same physical memory with Copy on Write, until one of the applications has to modify the contents. At that point, a new copy of the contents is made, and the writing process has its own copy.

It should be emphasized that this is not to say that applications are sharing memory in the sense that one application can write to it and another can read what the first one wrote; as long as applications are only going to read a piece of memory (for example, a code section), then the physical pages supporting that memory for the applications can be shared. Once the application needs to write to the memory (for example, in the form of a fixup), then that application must have a new physical page so that the modifications are not seen by other processes. The processes are no longer sharing the same physical pages.

Applications

When multiple instances of the same Windows-based application load, you may

notice that most, if not all, of their instance handles (hInstance) have the same value. In fact, almost all of the windows on the desktop have this value, which represents the base address where the application loaded in virtual memory.

Each of the instances of the same application running has its own protected virtual address space to run in. If each of these applications can load into its default base address, each will map to, and be able to share, the same physical pages in memory. Using Copy on Write, the system will allow these applications to share the same physical pages until one of the applications modifies a page. Then a copy is made in physical memory, and that process's virtual memory is fixed up to use the new physical page. If for some reason one of these instances cannot load in the desired base address, it will get its own physical pages. See the section on DLLs below for more explanation.

DLLs

Dynamic-link libraries (DLLs) are created with a default base address to load at. Assuming that multiple applications call the DLL, they will all try to load it within their own address space at that default virtual address. If they are all successful, they can all map that virtual address space to share the same physical pages.

However, if for some reason the DLL cannot be loaded within the process's address space at the default address, it will load the DLL elsewhere. The DLL must be copied into another physical address frame. The reason is that fixups for jump instructions in a DLL are written as specific locations within the DLL's pages. If the DLL can be loaded at the same base address for each process, the second to the nth process does not have to write that memory location for the jump. If a process cannot load the DLL at the specified base address, the locations written in the DLL's jumps will be different for this process. This forces the fixup to write a new location into the jump, and the Copy on Write will automatically force a new physical page.

Note that all references to data must be fixed up too. If this causes virtual memory of the code section to be updated, then the process will again go through the Copy on Write process. For example, if there are a great many places in the code section that make reference into data in a DLL, if the DLL cannot be loaded at its default location, the locations in the code section data in the DLL are referenced will have to be modified. If this is one of multiple instances of a process, these fixups must go through the copy on write process, and the virtual memory pages of the process's code section will not be able to map and share the same physical pages as the other instances. If there are a lot of references to the data in the DLL by the code section, this can essentially cause the entire code section to be copied to new physical pages.

POSIX

In POSIX, there is a fork() instruction that basically creates two copies of the same program. It is an expensive process for the system to copy the

address space of one process into another. Instead, under Windows NT, the system simply marks the parent's pages with Copy on Write. This way new physical frames are copied only if and when they are needed (have been modified). The system does not waste time or memory if all of the address space doesn't need to be copied. (For more information, see "Inside Windows NT" by Microsoft Press).

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMm

Copying Compressed Files

PSS ID Number: Q130331

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51

SUMMARY
=======

It is not possible to copy NTFS compressed files without uncompressing them. This functionality is not available in Windows NT versions 3.5 and 3.51; however, this feature may be included in a future version of Windows NT.

Compressed files are expanded via memory-mapped files. This minimizes the performance hit of expanding the file.

MORE INFORMATION
================

One of the new features found in Windows NT 3.51 is file compression. Files or directories can be compressed or decompressed by calling DeviceIoControl() with one of the following compression flags:

FSCTL_SET_COMPRESSION	Sets the compression state of a file or directory.
FSCTL_GET_COMPRESSION	Obtains the compression state of a file or directory.

Two additional FSCTL codes are documented in the Win32 SDK as "not implemented in this release." They are FSCTL_READ_COMPRESSION and FSCTL_WRITE_COMPRESSION. These additional FSCTL codes will be part of the functionality that will allow you to read and write files on an NTFS compressed drive without having to decompress them first. Again, this functionality may be included in a future release of Windows NT.

Additional reference words: 3.50 NTFS File Compression
KBCategory: kbprg kbusage
KBSubcategory: BseFileIo

Correct Use of Try/Finally

PSS ID Number: Q83670

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Try/finally, used correctly, helps to provide a robust application. However, if used incorrectly it can cause unnecessary overhead. any flow of control out of the try body of try/finally is an abnormal termination that can cause hundreds of instructions to be executed on an x86 system, and thousands on a MIPS machine, even if control leaves the try body via a control statement on the very last statement of the try body. The language definition states that control must leave the try body sequentially for normal termination to occur (that is, execution falls through the bottom of the try body).

The following sample demonstrates an incorrect use of try/finally:

```
/* Incorrect use of try/finally */

VOID
function (
   DWORD ... P1,
      .
   DWORD ... Pn
   )

{

   try {
       if (...) {
           .
           .
           return;
       }
           .
           .
   } finally {
      .
      .
   }
   return;
}
```

The overhead can be avoided in the above example by moving the return AFTER the end of the finally clause. The following provides more

detail on the correct use of try/finally.

MORE INFORMATION
================

Execution of a termination handler due to abnormal termination of a try body is expensive. Abnormal termination occurs when control leaves a try body in any way other than by falling through the bottom. Intentionally branching out of a try body is still an abnormal termination.

In the above example, abnormal termination of the try body occurs if the return in the middle of the try body is executed. If the predicate of the if is false, then extremely efficient execution of the finally clause occurs because this is not abnormal termination and the finally clause is called directly by inline code.

When abnormal termination occurs hundreds to thousands of instructions are executed because an unwind must be executed, which must search backward through frames to determine if any termination handlers should be called. On an x86 system, this executes the C run-time handler and examines the handler list. On a MIPS machine, this also causes the function table to be searched and the prologue of each intervening function to be executed backwards interpretively.

You should always avoid the execution of a termination handler as a result of the abnormal termination of a try body by a return, or other direct flow of control out of the try body. Abnormal termination occurs whenever control leaves the try body other than by falling through the bottom. This can occur because of a return, goto, continue, or break. It can also occur because of an exception, which presumably cannot be avoided.

In the above example, abnormal termination in the nonexception case can be eliminated easily as follows:

```
/* Correct use of try/finally */

VOID
function (
   DWORD ... P1,
      .
      .
   DWORD ... Pn
   )

{

   try {
       if (...) {
           .
           .
       } else {
           .
           .
```

```
        }
    } finally {
        .
        .
    }
    return;
}
```

Now both clauses of the if fall through to the termination handler in all but exceptional cases and execute the termination handler in the most efficient way. This also has the same logical execution as the previous sample.

In summary, the correct use of try/finally is a powerful method to help you write robust applications. Care should be taken to ensure the correct use of try/finally.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseExcept

CPU Quota Limits Not Enforced

PSS ID Number: Q100329

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY
=======

On page 88 of "Inside Windows NT," Table 4-1 indicates that a process object contains a quota limit for the maximum amount of processor time that the process can use.

This limit is not enforced in Windows NT versions 3.1 or 3.5x.

MORE INFORMATION
================

The key to understanding Windows NT thread scheduling and resultant application behavior is knowing the central algorithm used. This algorithm is very simple, and is the same one a number of other operating systems use. It is "run the highest priority thread ready." A list of ready threads or processes exists; it is often called the "dispatch queue" or "eligible queue." The queue entries are in order based on their individual priority. A hardware-driven real-time clock or interval timer will periodically interrupt, passing control to a device driver that calls the process or thread scheduler. The thread scheduler will take the highest priority entry from the queue and dispatch it to run.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseProcThrd

CreateFile() Using CONOUT$ or CONIN$

PSS ID Number: Q90088

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

If you attempt to open a console input or output handle by calling the CreateFile() function with the special CONIN$ or CONOUT$ filenames, this call will return INVALID_HANDLE_VALUE if you do not use the proper sharing attributes for the fdwShareMode parameter in your CreateFile() call. Be sure to use FILE_SHARE_READ when opening "CONIN$" and FILE_SHARE_WRITE when opening "CONOUT$".

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseFileio

CreateFileMapping() SEC_* Flags

PSS ID Number: Q108231

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The definition of CreateFileMapping() is as follows:

```
HANDLE CreateFileMapping(hFile, lpsa, fdwProtect, dwMaximumSizeHigh,
   dwMaximumSizeLow, lpszMapName)

HANDLE hFile;
LPSECURITY_ATTRIBUTES lpsa;
DWORD fdwProtect;
DWORD dwMaximumSizeHigh;
DWORD dwMaximumSizeLow;
LPCTSTR lpszMapName;
```

The following flags are four possible values for the parameter fdwProtect:

```
SEC_COMMIT
   All pages of a section are to be set to the commit state.
SEC_IMAGE
   The file specified for a section's file mapping is an executable
   image file.
SEC_NOCACHE
   All pages of a section are to be set as noncacheable.
SEC_RESERVE
   All pages of a section are to be set to the reserved state.
```

If none of these flags are specified, SEC_COMMIT is the default. This behaves the same way as MEM_COMMIT in VirtualAlloc().

MORE INFORMATION
================

Windows NT

The SEC_RESERVE flag is intended for file mappings that are backed by the paging file, and therefore use SEC_RESERVE only when hFile is -1. The pages are reserved just as they are when the MEM_RESERVE flag is used in VirtualAlloc(). The pages can be committed by subsequently using the VirtualAlloc() application programming interface (API), specifying MEM_COMMIT. Once committed, these pages cannot be decommitted.

The SEC_NOCACHE flag is intended for architectures that require various locking structures to be located in memory that is not ever fetched into the CPU cache. On x86 and MIPS machines, use of this flag just slows down the performance because the hardware keeps the cache coherent. Certain device drivers may require noncached data so that programs can write through to the physical memory. SEC_NOCACHE requires that either SEC_RESERVE or SEC_COMMIT be used in addition to SEC_NOCACHE.

The SEC_IMAGE flag indicates that the file handle points to an executable file, and it should be loaded as such. The mapping information and file protection are taken from the image file, and therefore no other options are allowed when SEC_IMAGE is used.

Windows 95

Under Windows NT, the Win32 loader simply sits on top of the memory mapped file subsystem, and so when the loader needs to load a PE executable, it simply calls down into the existing memory mapped file code. Therefore, it is extremely easy for to support SEC_IMAGE in CreateFileMapping() under Windows NT.

In Windows 95, the loader is more complex and the memory mapped files are simple and only support the bare minimum of functionality to make the existing MapViewOfFile() work. Therefore, Windows 95 does not support SEC_IMAGE. There is support for SEC_NOCACHE, SEC_RESERVE and SEC_COMMIT.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbprg
KBSubcategory: BseMm

Creating a Font for Use with the Console

PSS ID Number: Q105299

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

It is possible to use FontEdit to create a font that can be used by the console. The following must be true:

- The face name must be System, Terminal, or Courier
- The font size must be different from any of the other console fonts
- The font must be fixed pitch
- The font must not be italic

In addition, in the U.S. market, the font should support codepage 437.

Install the font from the Control Panel. After rebooting, the font will be available to the console.

An EnumFonts() call is made by the console during its initialization to determine what fonts are available. The console saves a set of one-to-one mappings between the font sizes listed and a set of LOGFONTs. The console never has direct knowledge of what file is used.

Additional reference words: 3.10 3.50
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Creating a Hidden MDI Child Window

PSS ID Number: Q70080

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

=======

Whenever Windows creates a new multiple-document interface (MDI) child window in response to a WM_MDICREATE message, it makes that child window visible.

The information below describes how to create a hidden MDI child window without causing an unattractive "flash" on the screen as the window is created visible and then hidden.

MORE INFORMATION

================

A code fragment such as the following can be used to create an invisible MDI child:

```
MDICREATESTRUCT mcs;            // structure to pass with WM_MDICREATE
HWND            hWndMDIClient;  // the MDI client window
HWND            hwnd;           // temporary window handle

      ...

// assume that we have already filled out the MDICREATESTRUCT...

// turn off redrawing in the MDI client window
SendMessage(hwndMDIClient, WM_SETREDRAW, FALSE, 0L);

/*
 * Create the MDI child. It will be created visible, but will not
 * be seen because redrawing to the MDI client has been disabled
*/
hwnd = (WORD)SendMessage(hwndMDIClient,
                         WM_MDICREATE,
                         0,
                         (LONG)(LPMDICREATESTRUCT)&mcs);

// hide the child
ShowWindow(hwnd, SW_HIDE);

// turn redrawing in the MDI client back on,
```

```
// and force an immediate update
SendMessage(hwndMDIClient, WM_SETREDRAW, TRUE, 0L);
InvalidateRect( hwndMDIClient, NULL, TRUE );
UpdateWindow( hwndMDIClient );
      ...
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMdi

Creating a List Box That Does Not Sort

PSS ID Number: Q68116

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

A Windows list box with the LBS_STANDARD style will sort the list of items into alphabetical order before displaying them in the control.

To create a list box that will not sort, you must remove the LBS_SORT bit from the window style. The following style specification removes this bit:

```
   (LBS_STANDARD | LBS_HASSTRINGS) & ~LBS_SORT
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Creating a List Box with No Vertical Scroll Bar

PSS ID Number: Q68115

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

A Windows list box with the LBS_STANDARD style will display a vertical scroll bar if there are more items in the list than can be displayed in the client area of the list box.

To create a list box that will not use a vertical scroll bar, you must remove the WS_VSCROLL bit from the window style. The following style specification removes this bit:

```
   (LBS_STANDARD | LBS_HASSTRINGS) & ~WS_VSCROLL
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Creating a List Box Without a Scroll Bar

PSS ID Number: Q11365

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

When LBS_STANDARD is used as follows

```
   LBS_NOTIFY | LBS_SORT | WS_BORDER | LBS_STANDARD
```

the following results (as defined in WINDOWS.H):

```
   LBS_STANDARD = #00A00003;
                  /* LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER */
```

To create a dialog box that contains a list box without the vertical scroll bar, use NOT WS_VSCROLL as the style for creating a list box control without a vertical scroll bar, as follows:

```
   (LBS_STANDARD & ~WS_VSCROLL)  // NOT WS_VSCROLL
```

Additional reference words: 3.00 3.10 3.50 4.00 95 listbox
KBCategory: kbui
KBSubcategory: UsrCtl

Creating a Logical Font with a Nonzero lfOrientation

PSS ID Number: Q104010

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

To create a font that writes in a direction other than left to right, an application should specify a nonzero lfEscapement in the LOGFONT structure that is passed to CreateFontIndirect(). This method works under Windows NT regardless of the graphics mode of the device context.

To create a font where the characters themselves are rotated, the application should specify a nonzero lfOrientation in the LOGFONT structure that is passed to CreateFontIndirect(). However, this setting is ignored in Windows NT unless the graphics mode is set to GM_ADVANCED.

Therefore, to successfully create a logical font with a nonzero lfOrientation, use

```
SetGraphicsMode( hDC, GM_ADVANCED )
```

to set the graphics mode of the device context to GM_ADVANCED.

MORE INFORMATION

The TTFONTS sample program is a good way to quickly and easily see the effects of the lfEscapement and lfOrientation fields. However, TTFONTS does not set the graphics mode of its test window HDC to GM_ADVANCED. As a result, the lfOrientation field apparently is ignored. It is easy to modify the DISPLAY.C module of TTFONTS in order to set the graphics mode of the window HDC to GM_ADVANCED.

Additional reference words: 3.10 3.50
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Creating a Multiple Line Message Box

PSS ID Number: Q67210

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

Message boxes are used to provide information to the user of an application. Error messages and warnings are also provided through message boxes. This article provides details on using message boxes in applications.

MORE INFORMATION
================

Message boxes are modal windows. When an application displays an application modal message box, which is the default message box type, the user cannot interact with any part of that application until the message box has been dismissed. However, the user may use the mouse or keyboard to activate another application and interact with it while the message box is displayed. Certain critical errors that may affect all of Windows are displayed in system modal message boxes. Windows will not perform any operations until the error condition is acknowledged and the system modal message box is dismissed.

There are times where it is necessary to display a long message in a message box. Windows does this when you start an MS-DOS-based application that uses graphics from inside an MS-DOS window. To break a message into many lines, insert a newline character into the message text. Here is a sample MessageBox() call:

```
   MessageBox(hWnd, "This is line 1.\nThis is line 2.", "App",
       MB_OK | MB_ICONQUESTION);
```

If the text of a message is too long for a single line, Windows will break the text into multiple lines.

System modal message boxes treat the newline character as any other. A newline character is displayed as a black block in the text. Because system modal message boxes are designed to work at all times, even under extremely low memory conditions, it does not provide the ability to display more than one line of text.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui

KBSubcategory: UsrDlgs

Creating a Nonblinking Caret

PSS ID Number: Q74607

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The Microsoft Windows graphical environment is designed to provide a blinking caret. However, using a timer and the SetCaretBlinkTime() function, an application can "trick" the caret into not blinking.

MORE INFORMATION

Although Windows is designed to blink the caret at a specified interval, a timer function and SetCaretBlinkTime() can be used to prevent Windows from turning the caret off by following these three steps:

1. Call SetCaretBlinkTime(10000), which instructs Windows to blink the caret every 10,000 milliseconds (10 seconds). This results in a "round-trip" time of 20 seconds to go from OFF to ON and back to OFF (or vice versa).

2. Create a timer, using SetTimer(), specifying a timer procedure and a 5,000 millisecond interval between timer ticks.

3. In the timer procedure, call SetCaretBlinkTime(10000). This resets the timer in Windows that controls the caret blink.

When an application implements this procedure, Windows never removes the caret from the screen, and the caret does not blink.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCrt

Creating a World SID

PSS ID Number: Q111543

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

The sample code below demonstrates how to create a World Security Identifier (SID). The World SID (S-1-1-0) is a group that includes all users. To determine if a SID (perhaps in an Access Control Entry) is the World SID, you must first create a World SID to compare it to. Once you have created a World SID, you can use the EqualSid() API (application programming interface) to determine equality.

Sample Code

```
PSID psidWorldSid;
SID_IDENTIFIER_AUTHORITY siaWorldSidAuthority =
     SECURITY_WORLD_SID_AUTHORITY;

psidWorldSid = (PSID)LocalAlloc(LPTR,GetSidLengthRequired( 1 ));

InitializeSid( psidWorldSid, &siaWorldSidAuthority, 1);
*(GetSidSubAuthority( psidWorldSid, 0)) = SECURITY_WORLD_RID;
```

Additional reference words: 3.10 and 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Creating Access Control Lists for Directories

PSS ID Number: Q115948

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

The discretionary access control list (DACL) for a directory usually differs from that of a file. When assigning security to a directory, you are often specifying both the security for the directory and the security for any contained files and directories.

A directory's ACL will normally contain at least two access control entries (ACE):

- An ACE for the directory itself and any subdirectories.
- An ACE for any files in the directory.

If an ACE is to apply to object in the directory (subdirectories and files), the ACE is marked as an OBJECT_INHERIT_ACE and/or a CONTAINER_INHERIT_ACE. (In this article, a container means a directory.)

For example, when you use File Manager to set the security on a directory to "Change (RWXD)(RWXD)," the directory's DACL contains the following two ACEs:

```
   ACE1 (applies to files in the directory)
      ACE flags:   INHERIT_ONLY_ACE | OBJECT_INHERIT_ACE
      Access Mask: DELETE | GENERIC_READ | GENERIC_WRITE |
                   GENERIC_EXECUTE

   ACE2 (applies to the directory and subdirectories)
      ACE flags:   CONTAINER_INHERIT_ACE
      Access Mask: DELETE | FILE_GENERIC_READ | FILE_GENERIC_WRITE |
                   FILE_GENERIC_EXECUTE
```

MORE INFORMATION

================

The ACE flags are part of the ACE header. The structure of an ACE header can be found in the online help by searching on "ACE_HEADER".

In the above example, ACE1 applies only to contained files through the INHERIT_ONLY_ACE flag. If INHERIT_ONLY_ACE is not specified in an ACE, the ACE applies only to the current container.

NOTE: Adding one of these ACEs to a directory does not change the security

for any contained files or directories. The ACEs are only copied to files and directories created after the ACEs have been added to the directory.

When adding your own security to files, it is easy to create a combination that File Manager does not recognize as a "standard" setting. This is shown in file manager as "special" security.

If you want to match the DACLs you create to those used by File Manager, you can set the security of a file or directory in File Manager and then check the DACLs and ACEs. A tool for this is provided as a sample called "Check_SD" in the Win32 SDK. Check_SD can be found in the Q_A\SAMPLES\CHECK_SD directory on the Win32 SDK CD.

REFERENCES
==========

- "Microsoft Win32 Programmer's Reference," Microsoft Corporation.

- "Microsoft Win32 SDK API Reference help file," Microsoft Corporation.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Creating and Using a Custom Caret

PSS ID Number: Q74514

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

In the Microsoft Windows graphical environment, creating a custom caret is simple. Windows has a series of caret control, creation, and deletion functions specifically designed to make manipulating the caret easy.

MORE INFORMATION
================

The caret is a shared system resource. Unlike brushes, pens, device contexts and such, but like the cursor, only one caret is available under Windows. Also, like the cursor, an application can define a custom shape for the caret.

The CreateCaret() function creates a custom caret. Its syntax is as follows:

```
   void CreateCaret(HWND hWnd, HBITMAP hBitmap,
                    int nWidth, int nHeight);
```

The caret shape can be a line, a block, or a bitmap specified as the hBitmap parameter. If the hBitmap parameter contains a valid handle [a bitmap handle returned from the CreateBitmap(), CreateDIBitmap(), or LoadBitmap() function], CreateCaret() ignores the values of its nWidth and nHeight parameters and uses the dimensions of the bitmap. If hBitmap is NULL, the caret is a solid block; if hBitmap is one, the caret is a gray block. The nWidth and nHeight parameters specify the caret size in logical units. If either nWidth or nHeight is zero, the caret width or height is set to the window-border width or height.

If an application uses a bitmap for the caret shape, the caret can be in color; unlike the cursor, the caret is not restricted to monochrome.

CreateCaret() automatically destroys the previous caret shape, if any, regardless of which window owns the caret. The new caret is initially hidden; call the ShowCaret() function to display the caret.

Because the caret is a shared resource, a window should create a caret

only when it has the input focus or is active. It should destroy the caret before it loses the input focus or becomes inactive. Only the window that owns the caret should move it, show it, hide it, or modify it in any way.

Other functions related to the caret are the following:

- SetCaretPos()
  This function moves the caret to the specified position (in logical coordinates).

- GetCaretPos()
  This function retrieves the caret's current position (in screen coordinates).

- ShowCaret()
  This function shows the caret on the display at the caret's current position. When shown, the caret flashes automatically. If the caret is not owned by the window specified in the call, the caret is not shown.

- HideCaret()
  This function hides the caret by removing it from the display screen. HideCaret() hides the caret only if the window handle specified in the call is the window that owns the caret. Hiding the caret does not destroy it.

  NOTE: Hiding the caret is cumulative; ShowCaret() must be called once for every call to HideCaret(). For example, if HideCaret() is called five times, ShowCaret() must be called five times for the caret to be shown.

- DestroyCaret()
  This function removes the caret from the screen, frees the caret from the current owner-window, and destroys the current shape of the caret. It destroys the caret only if the current task owns the caret. This call should be used in conjunction with CreateCaret(). DestroyCaret() does not free or destroy a bitmap used to define the caret shape.

- SetCaretBlinkTime()
  This function sets the caret blink rate. After the blink rate is set, it remains the same until the same window changes it again, another window changes it, another application changes it, or Windows is rebooted.

- GetCaretBlinkTime()
  This function returns the current caret blink rate.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCrt

Creating Autosized Tables with Windows Help

PSS ID Number: Q81233

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5 and 3.51

SUMMARY

A Windows Help file can contain a table to present information in a consistent manner. Help 3.1 supports "autosized" tables where the relative widths of the columns remain the same, even as the absolute size of the table changes. The absolute size is determined by the size of the Windows Help window.

MORE INFORMATION

To create an autosized table in the RTF file, create a centered table. Create the text for the table using the minimum possible width for each column. If the Help window is larger, the columns will be wider and Help will compute where the text wraps based on the available width. If the user sizes the Help window smaller than the table's authored size, Help will display a horizontal scroll bar.

In Word for Windows version 1.1, use the Table command on the Format menu to create a centered table. In Word for Windows 2.0, create a table, then choose Row Height from the Table menu and choose the Center button in the Alignment group.

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsHlp

Creating Instance Data in a Win32s DLL

PSS ID Number: Q109620

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

SUMMARY

The Win32 dynamic-link libraries (DLLs) that are running on Win32s use shared data by default. This means that any global data in the DLL is shared by all processes that use the DLL. Thread local storage (TLS) can be used to create instance data; that is, data in the DLL that is specific to each process.

The Win32s "Programmer's Reference" mentions that TLS can be used to create instance data, but provides no details. The sample code below shows the source for a DLL that uses instance data on both Win32 and Win32s. The sample code was built using Microsoft Visual C++, 32-bit edition.

If you use a development environment that does not have similar support for TLS, you should still be able to use the API (application programming interface) calls. The API calls for TLS are TlsAlloc, TlsGetValue, TlsSetValue, and TlsFree.

MORE INFORMATION

One reason for wanting to create instance data on Win32s is to create a DLL that behaves identically on Win32s and Win32 (although it introduces extra overhead on Windows NT and Windows 95). Another way to create a DLL that behaves identically on Win32s and Win32 is to share all of the data in the Win32-based DLL. For additional information, please see the following article(s) in the Microsoft Knowledge Base:

   ARTICLE-ID: Q109619
   TITLE   : Sharing All Data in a DLL

Sample Code

```
/* Compile options used: /LD /MD
*/

int __declspec(thread) nVar = 0;    // Variables should be initialized

int __declspec(dllexport) GetVar()
{
    return nVar;
}

void __declspec(dllexport) SetVar(int nNew)
```

```
{
    nVar = nNew;
}
```

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Creating Lines with a Nonstandard Pattern

PSS ID Number: Q34614

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The Microsoft Windows graphical environment provides six predefined pens for drawing dotted, dashed, and solid lines. However, an application cannot draw fine gray lines, such as those on a Microsoft Excel spreadsheet, with these pens. This article describes how to create such lines.

MORE INFORMATION
================

An application can use the LineDDA function to produce any type of patterned line. Based on the endpoints of a line, LineDDA calculates each point on the line and calls an application-defined callback function for each point. The callback function is free to use the calculated points in any manner desired. An application can draw a gray line similar to those used in Excel by calling the the SetPixel function in the callback function to draw every other point.

For example, the following code calculates all points on the line from coordinates (30, 40) to (100, 100). Then it calls the function pointed to by the lpfnLineProc variable with the points and the handle to a device context (hDC) as parameters:

```
LineDDA(30, 40, 100, 100, lpfnLineProc, (LPSTR)hDC);
```

For more information on this function, see pages 4-272 and 4-273 of the "Microsoft Windows Software Development Kit Reference, Volume 1" for Windows 3.0 or pages 568 and 569 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 2: Functions" for Windows 3.1. Charles Petzold's book "Programming Windows 3" (Microsoft Press, 1990) demonstrates using the LineDDA function in a programming example on pages 593 through 598.

The following code fragment draws 50 random Excel-style lines. Note that the LineProc function must be listed as an EXPORT in the module definition (DEF) file:

```
case WM_PAINT:
```

```
      {
      HDC hDC;
      int nIndex;
      PAINTSTRUCT ps;

      hDC = BeginPaint(hWnd, &ps);

      for (nIndex = 0; nIndex < 50; nIndex++)
         LineDDA(rand() / 110, rand() / 110, rand() / 110,
                 rand() / 110, lpfnLineProc, (LPSTR)hDC);

      EndPaint(hWnd, &ps);
      break;
      }

void FAR PASCAL LineProc(x, y, lpData)
short x, y;
LPSTR lpData;
{
   static short nTemp = 0;

   if (nTemp == 1)
      SetPixel((HDC)lpData, x, y, 0L);

   nTemp = (nTemp + 1) % 2;
}
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiDrw

Creating Windows in a Multithreaded Application

PSS ID Number: Q90975

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In a multithreaded application, any thread can call the CreateWindow() API to create a window. There are no restrictions on which thread(s) can create windows.

It is important to note that the message loop and window procedure for the window must be in the thread that created the window. If a different thread creates the window, the window won't get messages from DispatchMessage(), but will get messages from other sources. Therefore, the window will appear but won't show activation or repaint, cannot be moved, won't receive mouse messages, and so on.

MORE INFORMATION

Normally, windows created in different threads process input independently of each other. The windows have their own input states and the threads are not synchronized with eachother in regards to input processing.

In order to have threads to share input state, have one thread call AttachThreadInput() to have its input processing attached to another thread. What this means is that these two threads will use a Windows 3.1 style system queue. The threads will still have separate input, but they will take turns reading out of the same queue.

Creating a window can force an implicit AttachThreadInput(), when a parent window is created in one thread and the child window is being created in another thread. When windows are created (or set) in separate threads with a parent-child relationship, the input queues are attached.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Creating/Managing User Accounts Programmatically

PSS ID Number: Q119671

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

SUMMARY

Windows NT and the Windows NT Advanced Server use the same APIs that Microsoft LAN Manager uses to create and maintain user- and group-account information. For example, to create a new global group, use NetGroupAdd(). To create a new user, use NetUserAdd(). To add the user to the global group, use NetGroupAddUser(). Local groups are created by using NetLocalGroupAdd() and members are added to local groups by using NetLocalGroupAddMember().

MORE INFORMATION

The APIs NetGroupAdd(), NetUserAdd(), NetGroupAddUser(), NetLocalGroupAdd(), and NetLocalGroupAddMember() require access at the administrator or accounts-operator level to run successfully. Windows NT includes the following built-in groups:

- Administrators
- Power Users
- Users
- Guests

Members of the Administrators group can fully administer user accounts; only Administrators can assign user rights and access privileges for resources. Members of the Power Users group can create accounts only in the Power Users, Users, and Guests groups; they can also maintain and delete the accounts they create. However, a Power User can neither change nor delete an account in these groups if the account was created by someone else. A member of the Users group can create, maintain, and delete accounts in local groups that he or she has created. Guests can neither create nor delete accounts.

REFERENCES

In the Win32 SDK, version 3.1, the documentation for the ported LAN Manager APIs is available in the file LMAPI.HLP on the SDK CD. In the installed Win32 SDK, version 3.5, the ported LAN Manager APIs are documented in the Help file "Win32 API Reference".

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: NtwkLmapi

Critical Sections Versus Mutexes

PSS ID Number: Q105678

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

Critical sections and mutexes provide synchronization that is very similar, except that critical sections can be used only by the threads of a single process. There are two areas to consider when choosing which method to use within a single process:

1. Speed. The Synchronization overview says the following about critical sections:

   > ... critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization.

   Critical sections use a processor-specific test and set instruction to determine mutual exclusion.

2. Deadlock. The Synchronization overview says the following about mutexes:

   > If a thread terminates without releasing its ownership of a mutex object, the mutex is considered to be abandoned. A waiting thread can acquire ownership of an abandoned mutex, but the wait function's return value indicates that the mutex is abandoned.

   WaitForSingleObject() will return WAIT_ABANDONED for a mutex that has been abandoned. However, the resource that the mutex is protecting is left in an unknown state.

   There is no way to tell whether a critical section has been abandoned.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseSync

CS_SAVEBITS Class Style Bit

PSS ID Number: Q31073

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

If the CS_SAVEBITS style is included when registering a pop-up window, a bitmap copy of the screen image that the window will obscure is saved in memory when the window is displayed.

The bitmap is redisplayed at its original location and no WM_PAINT messages are sent to the obscured windows if the following is true when the window is removed from the display:

- The memory used by the saved bitmap has not been discarded.
- Other screen actions have not invalidated the image that has been stored.

As a general rule, this bit should not be set if the window will cover more than half the screen; a lot of memory is required to store color bitmaps.

The window will take longer to be displayed because memory needs to be allocated. The bitmap also needs to be copied over each time the window is shown.

Use should be restricted to small windows that come up and are then removed before much other screen activity takes place. Any memory calls that will discard all discardable memory, and any actions that take place "under" the window, will invalidate the bitmap.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrPnt

CTRL+C Exception Handling Under WinDbg

PSS ID Number: Q97858

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

SUMMARY

An exception is raised for CTRL+C only if the process is being debugged. The purpose is to make it convenient for the debugger to catch CTRL+C in console applications. For the purposes of this article, the debugger is assumed to be WinDbg.

MORE INFORMATION

When the console server detects a CTRL+C, it uses CreateRemoteThread() to create a thread in the client process to handle the event. This new thread then raises an exception IF AND ONLY IF the process is being debugged. At this point, the debugger either handles the exception or it continues the exception unhandled.

The "gh" command marks the exception as having been handled and continues the execution. The application does not notice the CTRL+C, with one exception: CTRL+C causes alertable waits to terminate. This is most noticable when executing:

```
while( (c = getchar()) != EOF )  - or -  while( gets(s) )
```

It is not possible to get the debugger to stop the wait from terminating.

The "gn" command marks an exception as unhandled and continues the execution. The handler list for the application is searched, as documented for SetConsoleCtrlHandler(). The handler is executed in the thread created by the console server.

After the exception is handled, the thread created to handle the event terminates. The debugger will not continue to execute the application if Go On Thread Termination is not enabled (from the Options menu, choose Debug, and select the Go On Thread Termination check box). The thread and process status indicate that the application is stopped at a debug event. As soon as the debugger is given a go command, the dead thread disappears and the application continues execution.

There are three cases where CTRL+C doesn't cause the program to stop executing (instead it causes a "page down"):

1. When CTRL+C is already being handled.

2. When the debugger is in the foreground and a source window has the

   focus (both must be true).

3. When the CTRL+C exception is disabled (through the Debugger
   Exceptions dialog box).

This follows the convention of the WordStar/Turbo C/Turbo Pascal editor commands.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsWindbg

CTYPE Macros Function Incorrectly

PSS ID Number: Q94323

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

When an application that is linked to CRTDLL.LIB is compiled without defining _MT and _DLL, the CTYPE.H family of macros will not operate correctly.

To define _MT and _DLL on the CL command line, just add the following to the command line:

```
-D_MT -D_DLL
```

By adding these defines, the CTYPE macros will be properly initialized.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsMisc

Custom Controls Must Use CS_DBLCLKS with Dialog Editor

PSS ID Number: Q71223

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

For a custom control to function properly with the Dialog Editor, the custom control window class must include the CS_DBLCLKS style.

If the custom control does not have the CS_DBLCLKS style, double-clicking the control in the Dialog Editor does not cause the custom control function to display its style dialog box. However, the control's style dialog box is still accessible from the Styles command on the Edit menu.

MORE INFORMATION
================

The Dialog Editor subclasses each control it creates and processes WM_LBUTTONDBLCLK messages. In response to this message, the custom control is asked to display its style dialog box.

If the custom control window class does not have the CS_DBLCLKS style, Windows does not send any WM_LBUTTONDBLCLK messages to the control. As a result, the Dialog Editor does not call the style dialog box function for the custom control and no dialog box appears.

NOTE: This article does not apply to the resource editor included with the Visual C++ development environment.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Customizing a Pop-Up Menu

PSS ID Number: Q12118

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The following are three methods to customize a pop-up menu:

1. Use the word SEPARATOR in a pop-up menu, to produce a horizontal bar.

2. Use the word MENUBREAK, to start the menus on another column.

3. Place the vertical bar symbol in the menu string to display a vertical bar on the menu.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMen

Customizing the FileOpen Common Dialog in Windows 95

PSS ID Number: Q125706

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

In Windows version 3.x, customizing the FileOpen common dialog meant providing a special hook function, and in most cases, a custom dialog box template. This custom dialog box template is created by modifying the standard FileOpen dialog box template used by COMMDLG, which was actually made available as part of the Windows versopm 3.1 SDK.

In Windows 95, the new dialog box templates will no longer be made available for modification. Instead, an application should provide a dialog template that includes only the items to be added to the standard dialog box. COMMDLG will then create this dialog as a child of the standard dialog box. Because it is a child, it must have the WS_CHILD style set.

Note that the Windows version 3.1 method of customizing common dialogs by modifying the dialog templates will still work for 16-bit applications.

MORE INFORMATION

================

Follow these steps to customize the FileOpen common dialog in Windows 95:

1. Create a dialog box template that will have all the controls you want to add to the FileOpen common dialog. Be sure to specify the styles:

   WS_CHILD | WS_VISIBLE | DS_3DLOOK | DS_CONTROL | WS_CLIPSIBLINGS

   WS_CHILD is specified because without it, the call to GetOpenFileName() fails. COMMDLG creates the dialog specified as a child of the standard FileOpen common dialog box. As a result, the hDlg passed to the application's hook function will be the child of the standard FileOpen dialog box. To get a handle to the standard dialog box from the hook function, call GetParent (hDlg).

   WS_CLIPSIBLINGS is specified so that overlapping controls paint properly.

   DS_3DLOOK is a new style for Windows 95 that gives the dialog box a nonbold font, and gives all the controls the 3D look.

   DS_CONTROL is another new style that among other things allows the user to tab between the controls of a dialog box to the controls

of a child dialog box. As mentioned above, the dialog template will be created as a child of the standard FileOpen common dialog box, so specifying this style will allow tabbing from the application-defined controls to the standard controls.

2. Include a static control in your dialog template, specifying a control ID of stc32. This control will serve as a placeholder for the standard controls.

   If there is no stc32 control specified, COMMDLG places all the new controls defined in your dialog template below the standard controls and looks at the size of the static control to attempt to fit all the standard controls in it. If it is not big enough, COMMDLG resizes this stc32 control to make room for the standard controls, and then repositions the new controls with respect to the resized stc32 control.

   Be sure to use the #include directive to include DLGS.H in your .RC file, as stc32 is defined in <dlgs.h>.

3. Initialize the Flags member of the OPENFILENAME structure to include the following flags:

      OFN_EXPLORER | OFN_ENABLETEMPLATE | OFN_ENABLEHOOK

   OFN_ENABLETEMPLATEHANDLE may be used instead of OFN_ENABLETEMPLATE if you want to specify a handle to a memory block containing a preloaded dialog box template.

4. If the OFN_ENABLETEMPLATE flag is set, specify the name of your application-defined template in the lpTemplateName field of the OPENFILENAME structure, and specify your application's instance handle in the hInstance field.

   If the OFN_ENABLETEMPLATEHANDLE flag is set, specify the handle to the memory block containing your dialog box template in the hInstance field of the OPENFILENAME structure.

5. Specify the address of a dialog box procedure associated with your dialog box in the lpfnHook field of the OPENFILENAME structure.

6. Process appropriate notifications and messages as a result of adding new controls.

REFERENCES
==========

Much of the information contained in this article is derived from the MSDN Technical Article entitled "Using the Common Dialogs Under Windows 95." Please refer to that article for more information.

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Dangers of Uninitialized Data Structures

PSS ID Number: Q74277

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

In general, all fields in structures passed to functions in the Microsoft Windows graphical environment should be initialized. If a field is not initialized, it may contain random data, which can cause unexpected behavior.

For example, before an application registers a window class, it must initialize the cbClsExtra and cbWndExtra fields of the WNDCLASS data structure. Windows allocates cbClsExtra bytes for the class, and cbWndExtra bytes for each window created using the class. If these fields contain large random values, the application may run out of memory quickly.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

DDE Error Message: Application Using DDE Did Not Respond

PSS ID Number: Q94955

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

DDEML displays a dialog box with the following error message when a terminate deadlock occurs, often caused by a DDE application not terminating correctly:

```
Application using DDE did not respond to the System's Exit command
```

MORE INFORMATION

A terminate deadlock situation occurs when DDEML times out while waiting for a responding terminate.

This message box appears when a DDEML application calls DdeUninitialize() with conversations still active. DdeUninitialize() posts WM_DDE_TERMINATE messages for each open conversation, and waits for a corresponding WM_DDE_TERMINATE for a set period of time. This time is actually set in the [DDEML] section of the WIN.INI file

```
[DDEML]
ShutdownTimeout= ?
ShutdownRetryTimeout=?
```

where both are defined as integers defaulting to 30000 milliseconds. These WIN.INI entries were purposely not documented to discourage people from setting them to some other value.

If DDEML does not receive a response within the set period of time, it brings up the message box to allow the user to choose to either quit, wait longer, or wait indefinitely. This was done to work around a problem in Windows 3.0 where the system locks up if an application attempts to post a message to a non-existent window, and to allow the user to save his work.

Additional reference words: 3.10 3.00 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

DdeInitialize(), DdeNameService(), APPCMD_FILTERINITS

PSS ID Number: Q108925

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

DdeInitialize() and DdeNameService() work as complimentary functions during initialization of a server application. When called for the first time to initialize a server application, DdeInitialize() will cause DDEML to append the APPCMD_FILTERINITS flag to the third parameter of the function call by default. Once this flag is set, client applications will not be able to connect to the server until it is reset. This flag is reset when the function DdeNameService() is called to register the service name with DDEML. Also, if the server wishes to remain anonymous, then DdeInitialize() must be called the second time to specifically turn off the APPCMD_FILTERINITS flag.

MORE INFORMATION

The DdeInitialize() function registers an application with DDEML. This function must be called by both the client and the server applications before calling any other DDEML function.

When the server application calls DdeInitialize() for the first time, DDEML appends the APPCMD_FILTERINITS flag to the third parameter of the function call, regardless of whether the application specifies this flag. This flag when used, will prevent DDEML from sending the XTYP_CONNECT and XTYP_WILDCONNECT transactions to the server application until the server has created its string handles and performed other application-specific initialization. The server application then calls DdeNameService() to register its service name with DDEML so that other client or server applications are notified of its existence. Calling DdeNameService() after the server has gone through the process of initialization turns off the APPCMD_FILTERINITS flag.

Some DDEML server applications might not want to register their names with DDEML because of various reasons (for example, the server application is a custom server application that wants to service particular clients, and thus wishes to remain anonymous to the rest of the system).

In special cases like this, an application may choose not to call the DdeNameService() function, because this function broadcasts the name of the server to all DDEML applications on the system. Not calling the

DdeNameService() function, however, causes the APPCMD_FILTERINIT flag not to be reset properly, thus keeping the server from getting any XTYP_CONNECT or XTYP_WILDCONNECT transactions even from its clients.

One other way to reset this flag is to call DdeInitialize() a second time, without specifying the APPCMD_FILTERINITS flag.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

DDEML Application-Instance IDs Are Thread Local

PSS ID Number: Q94091

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

When using the DDEML (Dynamic Data Exchange Management Library) libraries from a spawned thread, the application-instance ID that is returned in the lpidInst parameter of DdeInitialize is thread local.

Therefore, the application-instance ID cannot be used by any other thread that is spawned by the process, nor can it be inherited from the parent.

To use the DDEML libraries within a thread, it is necessary to make both the DdeInitialize call and to use the DdeUninitialize call from within the thread; otherwise, there is no way to terminate the DDEML session.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrDde

Dealing w/ Lengthy Processing in Service Control Handler

PSS ID Number: Q120557

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51

The service control handler function must return within 30 seconds. If it does not, the Service Control Manager will return the following error: "Error 2186 - The service is not responding to the control function".

If a service needs to do lengthy processing when the service is in the service control handler, it should create a secondary thread to perform the lengthy processing and then return. This prevents the service from tying up the service control handler thread.

If you are processing a SERVICE_CONTROL_STOP, you may wish to register a status of SERVICE_STOP_PENDING. The dwWaitHint should be at least 30 seconds. You can make the control panel applet wait for a long time if you send multiple SERVICE_STOP_PENDING states which update the dwCheckPoint and use a long dwWaitHint.

The system shutting down is another event that limits the service control handler. The dwCtrlCode parameter for the service control handler returns SERVICE_CONTROL_SHUTDOWN. A service then has approximately 20 seconds to perform cleanup tasks. If the tasks are not done, the system shuts down regardless if the service shutdown is complete. If the user has selected "restart", all processes will halt quickly. If instead the system is left in the "shutdown" state, the service processes continue to run.

If you need a longer time to shut down or earlier notification, consider using SetConsoleCtrlHandler() or SetProcessShutdownParameters() instead of using SERVICE_CONTROL_SHUTDOWN. This is the same mechanism that the Service Controller uses to get its notification.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseService

Debugging a Service

PSS ID Number: Q98890

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51

SUMMARY
=======

The steps in this article illustrate how to debug a service under Windows NT using WinDbg, MSVC 2.1+, and NTSD. WinDbg and NTSD both ship with the Win32 Software Development Kit (SDK). For illustration purposes, these procedures use the SERVICE sample, which is built with debugging information by default. This sample is located in:

Win32 SDK 3.5:
Mstools\Samples\Win32\Service

Win32 SDK 3.51 and 4.0:
Mstools\Samples\Win32\Winnt\Service

To debug a service, you no longer need to log in your service on the LocalSystem account. Your service can be logged on as LocalSystem, or you can also debug a service that logs on a domain\account as long as you are also interactively logged onto that account. For example, if you're logged on as domain\user, the service must be set to log onto the domain\user account in the startup info. The domain\user account should have admin privileges to the machine and set for "log on as a service" policy rights.

MORE INFORMATION
================

Preparation
---

1. Build the sample.

2. Install the Simple service with the following command:

   Win32 SDK 3.5 and later:
   simple - install

   You will receive a message indicating whether you were successful.

3. Use the Control Panel's Services application to start the Simple service. With the Win32 SDK 3.5+, you can also use:

   sc start simpleservice

   The SC.EXE is located in Mstools\Bin.

4. Use PView to get the process ID (PID) for the Simple service. For
   example, if PView shows the process as simple(0xD5), the PID is 0xD5.

5. If you're using SDK versions 3.1 or 3.5, convert the PID from
   hexadecimal to decimal. For example, 0xD5 is 213 in decimal. Later
   versions of nDbg use hexadecimal PIDs. Both MSVC 2.0+ and NTSD use
   hexadecimal PIDs.

Debugging a Service with WinDBG
---

1. At a command prompt, go to the directory containing the sample
   executable and type:

   start windbg

   to start WinDbg in its own command shell.

2. In WinDbg, on the File menu, click Open, and open the source file
   (Simple.c).

3. Set breakpoints at lines 223, 245, 256, and 271. The lines will change
   color at this point.

4. Open a command window in WinDbg and type

   .attach <PID>

   Note that the lines where breakpoints are set will have changed colors.

5. Type "g" (a go command) in the WinDbg command window or press F5 to
   restart after the thread that WinDbg uses to do the .attach terminates.

6. At the command prompt, start the client by typing:

   Win32 SDK 3.5 and later:
      client [-pipe <pipename>] [-string <string>]

   For example: client -pipe \\.\pipe\simple -string "Apollo13"

7. Press F5 (a go command) to debug the service. The breakpoint hit
   will be on line 245. Press F5 again to go to the next breakpoint.
   Keep pressing F5 until line 223 waits again for a client to
   connect. Try connecting another client and repeat the same steps.

Exiting WinDbg will kill the service, which must be restarted manually
with the Control Panel.

Debugging a Service with MSVC 2.1+
---

1. Follow steps 1-5 in the "Preparation" section of this article.

2. At a command prompt, go to the directory containing the sample and type:

   MSVC /P <PID>

   where <PID> is the Process ID value you retrieved in step 4.

3. Set breakpoints at lines 223, 245, 256, and 271. The lines will change
   color at this point.

4. At the command prompt, start the client by typing

   Win32 SDK 3.5 and later:
      client [-pipe <pipename>] [-string <string>]

   For example: client -pipe \\.\pipe\simple -string "Apollo13"

5. Press F5 (a go command) to debug the service. The breakpoint hit
   will be on line 245. Press F5 again to go to the next breakpoint.
   Keep pressing F5 until line 223 waits again for a client to
   connect. Try connecting another client and repeat the same steps.

Debugging a Service with NTSD
----

1. Compile using -Zi and -Od.

2. Link using debug:full and debugtype:coff.

3. Load the program into the debugger.

   ntsd simple -install

4. Use s+ to change to source mode.

5. Set breakpoints on the main and "service_main".

   bp servive_main

6. Type "v .<number>" to list source lines starting at <number>. For
   example, type the following:

   v .240

A running service may be debugged by invoking the debugger from a command
line. After obtaining the PID from PVIEWER you can invoke NTSD from a
command line. For example:

NTSD -p <PID>

before using NTSD make sure you have it installed properly and an NTSD
section in your Tools.ini file. An example of a NTSD section is:

```
[NTSD]
sxe: 3c
sxe: cc
sxe: av
sxe: et
```

```
sxe: ep
sxe: ld
$u0: VeryLongName
VerboseOutput:true
DebugChildren=TRUE
LazyLoad=TRUE
StopFirst=TRUE
StopOnProcessExit=TRUE
```

Piping Output

When a service is started under a debugger, it is not possible to send debug output directly to a window on the User's desktop. The output can either be sent to a kernel debugger or piped through remote.exe which can be found in the \Mstools\Samples\Sdktools\Remote directory.

NOTE: The "System Account" and the "LocalSystem Account" are the same account.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbhowto
KBSubcategory: TlsWindbg

Debugging a System-Wide Hook

PSS ID Number: Q102428

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Debugging a system-wide hook such as a journal hook must be done with the extreme caution. When an application installs such a hook, it effectively takes control of user input. In effect, this disables the interface with the debugger. For example, after installing a journal record hook, you must unhook the record hook when you want to allow the debugger to regain control.

It is not possible to use an interactive debugger to debug an actively installed journal hook using a single machine. It is possible to use a remote debugger, because one interface can be blocked (or recording) while the other one does the debugging.

MORE INFORMATION

================

System-wide input hook procedures can be thought of as being in three possible states:

   unhooked (not installed)
   suspended
   hooked (installed)

In the unhooked state, the procedure imposes no control over user input. In the hooked state, all user input specifically defined to be handled by this hook passes through this procedure. In the suspended state, all user input specifically defined to be handled by this hook is completely blocked.

In the case of a journal record hook, the suspended state can be achieved when a breakpoint is reached within the hook procedure. When this happens, all user input (system wide, that is) in the form of mouse and keyboard input is blocked, and thus you cannot interact with the debugger or any other application as you normally would. Fortunately, when the user presses the CTRL+ESC or the CTRL+ALT+DEL key combinations, all system-wide hooks are automatically unhooked, returning the system to the unhooked state.

Once this has occurred, it is likely that the application with the journal hook is now in a undefined state (because it had the hook pulled out from underneath it, so to speak). Fortunately, the system will send all applications the WM_CANCELJOURNAL message to indicate that it has removed the hook. A well behaved application can intercept this message and adjust

its state accordingly.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrHks

Debugging Applications Under Win32s

PSS ID Number: Q102430

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

SUMMARY
=======

To start debugging a Win32-based application, make sure that it runs correctly under Windows NT. Use either WinDbg or NTSD to track down any problems.

Then, install the debugging libraries for Windows 3.1 and the debug version of Win32s. Hook the machine to another machine running a terminal emulator and watch for any warnings that are issued. Be sure to select the Win32sDebug flags carefully--selecting too many will generate more information than you may care to see; selecting too few may cause you to miss important information and warnings.

If you need to debug on Win32s, there are currently two options:

- Use wdeb386 (note that this method is very tricky).

- Use remote WinDbg (WDBG32S.EXE) if you are not familiar with using a kernel debugger. This method requires two machines: a Win32s machine to run the application and the remote debugger and a Windows NT machine to run WinDbg.

If you have Microsoft Visual C++ 32-bit Edition version 1.0, CodeView for Win32s is an additional option. CodeView for Win32s is a user-level debugger; remote debugging is not necessary, and therefore CodeView for Win32s does not require a second machine. CVW32S does not come with Visual C++ 2.0 and later. You can still use CVW32S.EXE with later versions of Visual C++, if you link with /PDB:none and /INCREMENTAL:no and if you do not use new features such as templates or C++ exception handling.

MORE INFORMATION
================

When performing remote debugging, make sure that the cable is set up exactly as specified in the Win32s Programmer's Reference. The remote WinDbg does not support software flow control, so it is very important that the hardware flow control is set up properly. If it is not set up correctly, you will have problems as the buffers overflow.

WinDbg supports XON/XOFF (software) flow control, which means that the standard 3-wire cable can now be used, although the default is still hardware handshaking (5-wire cable). To enable XON/XOFF, you must specify the XON flag in the serial transport parameters on both WinDbg and remote WinDbg.

To enable XON/XOFF in the remote WinDbg:

1. Select Options to bring up the Transport dynamic-link library (DLL) dialog box.

2. Select the serial transport and make any needed modifications to the communications port or baud rate parameters.

3. Place the XON flag at the end of the Parameters box. For example, "COM1:19200 XON". Note that the space is needed.

To enable XON/XOFF on WinDbg:

1. Select Options/Debug DLLs.

2. Select the proper serial transport layer.

3. Choose the Change button.

4. Add XON to the end of the Parameters line: "COM1:19200 XON".

It is very important that both sides of the debugger use the same protocol. If they do not, both debuggers will probably hang. Also, the remote debugging environment requires that binaries be located on the same drive/directory on both the development and target systems. For example, if WIN32APP.EXE is built from sources in a C:\DEV\WIN32APP directory, the binary should be located in this directory on both systems. If you build your source files by specifying fully qualified paths for the compiler, the compiler will place this information with the debug records which will allow WinDbg to automatically locate the appropriate source files.

For additional information on remote debugging, please see the "Win32s Programmer's Reference" which is included with the Win32 SDK.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Debugging Console Apps Using Redirection

PSS ID Number: Q102351

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SUMMARY

To redirect the standard input (STDIN) for a console application named APP.EXE from a file named INPUT.TXT, the following syntax is used:

```
app < input.txt
```

However, the following syntax will not work when attempting to debug this application using WinDbg with STDIN redirected:

```
windbg app < input.txt
```

To debug the application as desired, use

```
windbg cmd /c "app < input.txt"
```

MORE INFORMATION

This will allow WinDbg to debug whatever goes on in the cmd window. A dialog box will be displayed that says "No symbolic Info for Debuggee." This message refers to CMD.EXE; dismiss this dialog box. When the child process (APP.EXE) is started, the command window will read "Stopped at program entry point." To continue, type "g" at the command window. Note that APP.EXE will begin executing, then you can open the source file and set breakpoints.

This technique is also useful when debugging an application that behaves differently when run with a debugger than it does when it is run in the command window.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsWindbg

Debugging DLLs Using WinDbg

PSS ID Number: Q97908

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SUMMARY

This article describes the process of debugging dynamic-link libraries (DLLs) under WinDbg. As a further example, debugging File Manager extensions under Windows NT is discussed in the "More Information" section in this article.

MORE INFORMATION

The application and the DLL must be built with certain compiler and linker switches so that debugging information is included. These switches can be found in the $(cdebug) and $(ldebug) macros, respectively, which are defined in NTWIN32.MAK.

NOTE: It is important to disable optimization with -Od or locals will not be available in the locals window and line numbers may not match the source.

The application is loaded into WinDbg either by specifying "windbg <filename>" on the command prompt or by starting WinDbg from the program group and specifying <filename> in the Program Open dialog box (from the Program menu, choose Open). Note that <filename> is the name of the application, not the DLL. It is not necessary to specify the name of the DLL to be debugged.

The DLL is loaded either when execution of the application begins or dynamically through a call to LoadLibrary(). In the first case, simply press F8 to begin execution. All DLLs and symbolic information are loaded. To trace through the DLL code, breakpoints can be set in the DLL using a variety of methods:

- From the Debug menu, choose Breakpoints. The dialog box is Program Open.

  -or-

- Open the source file and use F9 or the "hand" button on the toolbar.

  -or-

- Go to the Command window and type:

      bp[#] <Options>

```
   <Options>:

      addr          break at address
      @line         break at line
```

In the case that the DLL is dynamically loaded, pressing F8 causes all other DLLs and symbolic information to load. The same methods described above can be used to set breakpoints; however, the user will get a dialog box indicating that the breakpoint was not instantiated. After the call to LoadLibrary() has been executed, all breakpoints are instantiated (it is possible to note the color change if the DLL source window is open) and will behave as expected.

To set a breakpoint in a DLL that is not loaded, specify the context when setting the breakpoint. The syntax for a context specifier is:

```
   {proc, module, exe}addr

   -or-

   {proc, module, exe}@line
```

Example: {func, module.c, app.exe}0x50987. The first two parameters are optional, so {,,app.exe}0x50987 or {,,app.exe}func could be used instead.

For example, assume that we are trying to debug a File Manager extension under Windows NT that has been built with full debugging information. The procedure to debug the extension is as follows:

1. Open a Command window.
2. Start WinDbg WINFILE.
3. Set a breakpoint on FmExtensionProc().
4. At the Command window, type "g" and press ENTER. The debugger will continue executing the program form the point where it stopped (which could be from the beginning, at the breakpoint, and so on).

WinDbg will start WINFILE and when FmExtensionProc() is executed, WinDbg will break into the WINFILE process.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsWindbg

Debugging OLE 2.0 Applications Under Win32s

PSS ID Number: Q123812

The information in this article applies to:

- Microsoft Win32s version 1.2

SUMMARY

The following are available to help you debug your OLE 2.0 applications under Win32s:

- Debug versions of the OLE DLLs, included with Win32s. (See the Win32s Programmer's Reference for more information on the debug DLLs.)

- Failure/trace messages.

- The OLE SDK for Win32s, version 1.2, included on the Microsoft Developer Network (MSDN) CD.

For information on a utility that will convert OLE error codes into error message, please see the following article in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q122957
TITLE     : SAMPLE: DECODE32: OLE Error Code Decoder Tool
```

MORE INFORMATION

The debug version of OLE can send diagnostic information to the debug terminal. To enable this feature, include the following lines in the SYSTEM.INI:

```
[Win32sDbg]
ole20str=xxxxx
ole20str16=yyy
```

Use ole20str for 32-bit OLE and ole20str16 for 16-bit OLE. Set them to a combination of the following letters (case sensitive):

```
f - Failure message, kind of asserts.
v - Verbose. General purpose messages.
1 - Trace special translation activity for 32/16 interoperability.
i - Trace initialization of OLE.
t - Trace termination and cleanup of OLE.
```

The following tools are contained in the OLE SDK for Win32s, version 1.2:

- DFVIEW - Show the content of storage files.

- LRPCSPY - Monitor LRPC messages sent by 16-bit OLE applications (does

not require Win32s).

- RPCSPY32 - Monitor LRPC messages from both 16-bit and 32-bit OLE applications.

- DOBJVIEW and DOBJVW32 - View objects placed on the clipboard as well as objects transferred by drag and drop.

- IROTVIEW and IROTVW32 - Display the contents of the OLE running object table (ROT).

- OLE2VIEW and OLE2VW32 - Identify objects, interfaces, inproc and local servers, registration database entries, and so on.

Additional reference words: 1.20
KBCategory: kbole kbprg
KBSubcategory: W32s

Debugging the Win32 Subsystem

PSS ID Number: Q105677

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

The instructions on page 1-18 of Part II of the Win32 "Programmer's Guide" included with the Win32 Device Driver Kit (DDK) says to use NTSD -d -p -1 to attach to the Win32 subsystem process and enable debugging of its user-mode drivers. This results in the error:

```
NTSD: cannot debug PID -1
error = 5
```

To enable this procedure to work properly, change the GlobalFlag value under:

```
HKEY_LOCAL_MACHINE\
            SYSTEM\
 CurrentControlSet\
           Control\
   Session Manager
```

Remove the flag 0x00080000 from 0x211a0000 to make it 0x21120000. The 0x00080000 flag disables the ability to debug CSRSS.EXE (the client server run time subsystem), which is specified by the "-p -1" parameter.

It is also possible to debug CSRSS using "WinDbgRm -c -p-1" instead of NTSD. Make sure that WinDbgRm defaults to debugging using TLPIPE.DLL as its transport layer, then run "windbgrm -c -p-1" on the debuggee.

On the debugger machine, make sure that CSRSS.EXE and any dynamic-link libraries (DLLs) that you are debugging in association with it are in the same directory, and run WinDbg. To set the transport DLL, choose Debug from the Options menu, choose Transport DLLs, and set the transport DLL to TLPIPE. Set the host name entries to be the machine name of the debuggee.

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsMisc

Debugging Universal Thunks

PSS ID Number: Q105756

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

The general recommendation for an application targeted for Win32s is to debug it under Windows NT, then make sure that the application works under Win32s. However, Universal Thunks are not supported on Windows NT, so it is not possible to debug Win32-based applications that use the Universal Thunk in this manner.

To debug across the Universal Thunk, you can use WDEB386, which is available with the Windows 3.1 Software Development Kit (SDK). If you are not familiar with WDEB386, you may find it simpler to use other methods. In that case, be sure to install the debug version of Windows 3.1 and the debug version of Win32s and enable suitable notifications for Win32s (unimplemented functions and messages, verbose, and so forth). You may find OutputDebugString() useful for displaying extra information.

For more information on WDEB386, please see the Knowledge Base article "Tips On Installing WDEB386." For information on installing the debug version of Windows, please see your Windows SDK documentation.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Default Attributes for Console Windows

PSS ID Number: Q90837

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

Console attributes (screen fonts, screen colors, insert/overstrike, etc.) are stored by console title within each user's profile. When a profile for a new window is not found, the .DEFAULT configuration is used.

MORE INFORMATION
================

You can change the start font, screen colors, pop up colors, and insert/overstrike defaults using REGEDT32. Under the key HKEY_USERS, there is a .DEFAULT along with an entry for each user who has an account on that machine. Select DEFAULT\Console\Configuration. Use Edit.AddValue to add FontSize as a REG_DWORD to change the default font, Add ScreenColors and PopupColors as REG_DWORDs to change those defaults. To reset the console to be in insert mode rather than overstrike mode, add InsertMode as REG_SZ and set it to ON.

To get the right settings for the font size and colors you should first set your MS-DOS Prompt window font size and colors. Look up Console\MS-DOS Prompt\Configuration under your account and write down the values for the keys you need to add. Then go back to DEFAULT\Console\Configuration and add those values.

The DEFAULT configuration is read when the user chooses the Command prompt. However, if the user chooses to run a command shell via the File Manager (selecting CMD.EXE and choosing Run from the File menu), the DEFAULT configuration will not be read out of the registry.

WARNING:

RegEdit is a very powerful utility that facilitates directly changing the Registry Database. Using RegEdit incorrectly can cause serious problems, including hard disk corruption. It may be necessary to reinstall the software to correct some problems. Microsoft does not support changes made with RegEdit. Use this tool at your own risk.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseCon

Default Edit Control Entry Validation Done by Windows

PSS ID Number: Q74266

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Under the Microsoft Windows graphical environment, multiline and single-line edit controls do not accept characters with virtual key code values less than 0x20. The two exceptions are the TAB and ENTER keys; users can enter these characters only in a multiline edit control.

If an application creates an edit control with the ES_LOWERCASE or ES_UPPERCASE style, text entry is converted into the specified case.

If an application creates an edit control with the ES_OEMCONVERT style, the text is converted from the ANSI character set to the OEM character set and then back to ANSI for display in the control.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Default Stack in Win32-Based Applications

PSS ID Number: Q97786

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

By default, space is reserved for applications in the following manner:

1 megabyte (MB) reserved (total virtual address space for the stack)

1 page committed (total physical memory allocated when stack is created)

Note: The -stack linker option can be used to modify both of these values.

The default stack size is taken from the process default reserve stack size.

The operating system will grow the stack as needed by committing 1 page blocks (4K on an x86 machine) out of the reserved stack memory. Once all of the reserved memory has been committed, Windows NT will attempt to continue to grow the stack into the memory adjacent to the memory reserved for the stack, as shown in the following example on an x86 machine:

```
|<--- Total 1 MB for stack --->|<--- Adjacent memory --->|
-------------------------------------------------------
|       |                      |                        |
|  4K   |   1020K  ...         |       ...              |
|       |                      |                        |
-------------------------------------------------------
```

However, once the stack grows to the point that the adjacent area is not free (and this may happen as soon as the reserved 1 MB has been committed), the stack cannot grow any farther. Therefore, it is very risky to rely on this memory being free. Applications should take care to reserve all the memory that will be needed by increasing the amount of memory reserved for the stack.

In other cases, it may be desirable to reduce the amount of memory reserved for the stack.

The /STACK option in the linker and the STACKSIZE statement in the DEF file can be used to change both the amount of reserved memory and the amount of committed memory. The syntax for each method is shown below:

```
/STACK:[reserve][,commit]
```

```
   STACKSIZE [reserve][,commit]
```

MORE INFORMATION
================

Each new thread gets its own stack space of committed and reserved memory. CreateThread() has a stacksize parameter, which is the commit size. If a new size is not specified in the CreateThread() call, the new thread takes on the same stack size as the thread that created it, whether that be the default value, a value defined in the DEF file, or by the linker switch. If the commit size specified is larger than the default process stack size, the stack size is set to the commit size. When specifying a stack size of 0, the commit size is taken from the process default commit.

The system handles committing more reserved stack space when needed, but cannot reserve or commit more than the total amount initially reserved (or committed if no additional is reserved). Remember that the only resource consumed by reserving space is addresses in your process. No memory or pagefile space is allocated. When the memory is actually committed, both memory and pagefile resources are allocated. There is no harm in reserving a large area if it might be needed.

As always, automatic variables are placed on the stack. All other static data is located in the process address space. Because they are static, they do not need to be managed like heap memory.

Note that under Win32s 1.2 and earlier, stacks are limited to a maximum of 128K (this limit has been increased with Win32s 1.25a). The same stack is used on the 16-bit side of a Universal Thunk (UT). A 16:16 pointer is created and it points to the top of the 32-bit stack. The selector base is set in such a way that the 16-bit code is allocated at least an 8K stack.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMm

Default/Private Dialog Classes, Procedures, DefDlgProc

PSS ID Number: Q68566

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The information below explains the differences between default and private dialog classes, their associated dialog procedures, and using the DefDlgProc() function.

This information is organized as a comparison between private and default dialog classes, covering class registration, dialog templates, dialog creation, and message processing.

Note that the source to the DefDlgProc() function is provided with the Windows Software Development Kit (SDK) version 3.0. The code is supplied on the Sample Source 2 disk in the DEFDLG.C file. By default, DEFDLG.C is placed into the \SAMPLES\DEFPROCS directory if the SDK installation program copies the sample source code.

There are many functions and macros used in DefDlgProc() that are internal to Windows and cannot be used by applications. No additional information is available on these functions and macros.

MORE INFORMATION

All dialog classes are window classes, just as all dialog boxes are windows. All dialog classes must declare at least DLGWINDOWEXTRA in the cbWndExtra field of the WNDCLASS structure before the dialog class is registered. The Windows Dialog Manager uses this area to store special information for dialog boxes.

The default dialog class is registered by Windows at startup. The window procedure for this class is known as DefDlgProc(), which is located in Windows's USER module. DefDlgProc() calls the application-provided dialog function, which returns TRUE if it processes a message completely or FALSE if DefDlgProc should process the message further.

If an application registers a private dialog class, it provides a window procedure for the dialog box. The window procedure is the same as that for any other application window and returns a LONG value. Messages that are not processed by this window function are passed to DefDlgProc().

Dialog Class Registration

Windows registers the default dialog class, which is represented by the value 0x8002. Windows uses this class when an application creates a dialog box using the DialogBox() or CreateDialog() functions, but specifies no class in the dialog resource template.

To use a private dialog class, the application must specify the fields of a WNDCLASS structure and call RegisterClass(). This is the same procedure that Windows uses to register the default dialog class.

In either case, the value in the cbWndExtra field of the WNDCLASS structure must contain a value of at least DLGWINDOWEXTRA. These bytes are used as storage space for dialog-box specific information, such as which control has the focus and which button is the default.

When a dialog class is registered, the lpfnWndProc field of the WNDCLASS structure must contain a function pointer. For the default dialog class, this field points to DefDlgProc(). For a private class, the field points to application-supplied procedure that returns a LONG (as does a normal window procedure) and passes all unprocessed messages to DefDlgProc().

Dialog Templates

Resource scripts are almost identical whether used with a default or a private dialog class. Dialog boxes using a private class must use the CLASS statement in the dialog template. The name given in the CLASS statement must match the name of class that exists (is registered) when the dialog box is created.

Dialog Creation and the lpfnDlgFunc Parameter

Applications create dialog boxes using the function DialogBox(), CreateDialog(), or one of the variant functions such as DialogBoxIndirect(). The complete list of functions is found on page 1-43 of the "Microsoft Windows Software Development Kit Reference Volume 1."

All dialog box creation calls take a parameter called lpfnDlgFunc, which can either be NULL or the procedure instance address of the dialog box function returned from MakeProcInstance(). When the application specifies a private dialog class and sets lpfnDlgFunc to a procedure instance address, the application processes each message for the dialog box twice. The message processing proceeds as follows:

1. Windows calls the dialog class procedure to process the message. To process a message in the default manner, this procedure calls

   DefDlgProc().

2. DefDlgProc() calls the procedure specified in the dialog box creation
   call.

The procedure specified in lpfnDlgFunc must be designed very carefully. When it processes a message, it returns TRUE or FALSE and does not call DefDlgProc(). These requirements are the same as for any other dialog procedure.

Using a dialog procedure in conjunction with a private dialog class can be very useful. Processing for the private dialog class can be generic and apply to a number of dialog boxes. Code in the dialog procedure is specific to the particular instance of the private dialog class.

Dialog Message Processing

In dialog boxes with the default class, the application provides a callback dialog function that returns TRUE or FALSE, depending on whether or not the message was processed. As mentioned above, DefDlgProc(), which is the window procedure for the default dialog class, calls the application's dialog function and uses the return value to determine whether it should continue processing the message.

In dialog boxes of a private class, Windows sends all messages to the application-provided window procedure. The procedure either processes the message like any other window procedure or passes it to DefDlgProc(). DefDlgProc() processes dialog-specific functions and passes any other messages to DefWindowProc() for processing.

Some messages are sent only to the application-supplied procedure specified in the call to CreateDialog() or DialogBox(). Two examples of functions that Windows does not send to the private dialog class function are WM_INITDIALOG and WM_SETFONT.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Defining Private Messages for Application Use

PSS ID Number: Q86835

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In the Microsoft Windows environment, an application can define a private message for its own use without calling the RegisterWindowMessage API. Message numbers between 0x8000 and 0xBFFF are reserved for this purpose.

For Windows NT and Windows 95, the system defines a new message WM_APP (value 0x8000). Applications can make use of the range WM_APP through 0xBFFF for private messages without conflict. The only requirement is that the .EXE file must be marked version 4.0 (use the linker switch /subsystem:windows,4.0). Windows NT 3.5 and 3.51 and Windows 95 will run applications marked version 4.0.

MORE INFORMATION

The documentation for the WM_USER message lists four ranges of message numbers as follows:

Message Number	Description
0 through WM_USER-1	Messages reserved for use by Windows.
WM_USER through 0x7FFF	Integer messages for use by private window classes.
0x8000 through 0xBFFF	Messages reserved for use by Windows.
0xC000 through 0xFFFF	String messages for use by applications.
Greater than 0xFFFF	Reserved by Windows for future use.

When an application subclasses a predefined Windows control or provides a special message in its dialog box procedure, it cannot use a WM_USER+x message to define a new message because the predefined controls use some WM_USER+x messages internally. It was necessary to use the RegisterWindowMessage function to retrieve a unique message number between 0xC000 and 0xFFFF.

To avoid this inconvenience, messages between 0x8000 and 0xBFFF were redefined to make them available to an application. Messages in this range do not conflict with any other messages in the system.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrMsg

Definition of a Protected Server

PSS ID Number: Q102447

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

The Win32 application programming interface (API) reference briefly discusses creating a "protected server" that assigns security to private objects. This article explains the concept of a protected server" and its relationship to private objects.

MORE INFORMATION
================

A protected server is an application that provides services to clients. These services could be as simple as saving and retrieving information from a database while issuing security checks to verify that the client has proper access.

A private object is an application-defined data structure that both the client and server recognize. Private objects are not registered with nor recognized by the Windows NT operating system; they are entirely application-defined.

It is not uncommon for security to be assigned to private objects in a protected server's database. For example, when a client asks the server to create a new object in the database, the server could use the CreatePrivateObjectSecurity() Win32 API to create a security descriptor (SD) for the new private object. The server would then store the SD with the private object in the database. It is important to note that there is nothing in the SD that associates it with the private object. Instead, it is up to the protected server to maintain that association in the private object or in the database. It is likely that the private object and the associated SD would be stored together in a single database record.

A protected server application is responsible for checking a client's access before providing information. For example, when a client asks the server to retrieve some data, the server would go out and locate the record (which would contain the private object and SD) and bring a copy of the SD into memory. It would then call the AccessCheck() Win32 API passing the SD, the clients access token, and the desired access mask. AccessCheck() will check the client's access against the object's SD to determine if access is permitted. Depending on the result of AccessCheck(), the protected server would either provide the requested information or deny access.

In conclusion, a protected server is a application that performs operations on private objects that are entirely user defined. The protected server is

responsible for associating security descriptors to those objects and must take the steps necessary to verify a client's access.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Definitions for LDLLHandler and Other Advanced Help Features

PSS ID Number: Q150999

The information in this article applies to:

- Microsoft Win32 Software Development Kit for
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

- Microsoft Win16 Software Development Kit for
  - Microsoft Windows version 3.1

SUMMARY

This article provides the definitions necessary to use advanced Windows Help features such as embedded windows, LDLLHandler, and the internal functions exposed by Windows Help. This information is currently not documented in any of the shipping SDK documentation.

MORE INFORMATION

```
   /* Macro error structure and error code definitions */

   #define wMACRO_ERROR    128     /* Maximum error msg length */

   typedef struct {
     WORD  fwFlags;        /* indicates how error will be handled */
     WORD  wError;         /* Macro error code */
     /* Error message (if wError == wMERR_MESSAGE) */
     char  rgchError[wMACRO_ERROR];
   } ME, NEAR *PME,  FAR *QME;

   /* Macro error codes */
   #define wMERR_NONE      0       /* No error */
   #define wMERR_MEMORY    1       /* Out of memory (local) */
   #define wMERR_PARAM     2       /* Invalid parameter passed */
   #define wMERR_FILE      3       /* Invalid file parameter */
   #define wMERR_ERROR     4       /* General macro error */
   #define wMERR_MESSAGE   5       /* Macro error with message */

   /* Flags constants indicating how error may be handled */
   #define fwMERR_ABORT    0x0001  /* Allow the "abort" option */
   #define fwMERR_CONTINUE 0x0002  /* Allow the "continue" option */
   #define fwMERR_RETRY    0x0004  /* Allow the "retry" option */

   /* Classes of messages that may be sent to DLLs */
   #define DC_NOMSG     0x00
   #define DC_MINMAX    0x01
   #define DC_INITTERM  0x02
   #define DC_JUMP      0x04
   #define DC_ACTIVATE  0x08
```

```
#define DC_CALLBACKS 0x10

/* Messages sent to DLLs */
#define DW_NOTUSED   0
#define DW_WHATMSG   1
#define DW_MINMAX    2
#define DW_SIZE      3
#define DW_INIT      4
#define DW_TERM      5
#define DW_STARTJUMP 6
#define DW_ENDJUMP   7
#define DW_CHGFILE   8
#define DW_ACTIVATE  9
#define DW_CALLBACKS 10

/* Embedded Window messages */
#define EWM_RENDER          0x706A
#define EWM_QUERYSIZE       0x706B
#define EWM_ASKPALETTE      0x706C
#define EWM_FINDNEWPALETTE  0x706D

/* Embedded Window structure */
typedef struct tagCreateInfo {
  short   idMajVersion;
  short   idMinVersion;
  LPSTR   szFileName;   /* Current Help file */
  LPSTR   szAuthorData; /* Text passed by the author */
  HANDLE  hfs;          /* Handle to the current file system */
  DWORD   coFore;       /* Foreground color for this topic */
  DWORD   coBack;       /* Background color for this topic */
} EWDATA, FAR *QCI;

/* Embedded window rendering info */
typedef struct tagRenderInfo {
  RECT  rc;
  HDC   hdc;
} RENDERINFO, FAR * QRI;

/* file mode flags */

#define fFSReadOnly       (BYTE)0x01
#define fFSOpenReadOnly   (BYTE)0x02

#define fFSReadWrite      (BYTE)0x00
#define fFSOpenReadWrite  (BYTE)0x00

/* seek origins */
#define wFSSeekSet      0
#define wFSSeekCur      1
#define wFSSeekEnd      2

/* low level info options */
#define wLLSameFid    0
#define wLLDupFid     1
#define wLLNewFid     2
```

```
/* Return codes (help file system) */
#define rcSuccess       0
#define rcFailure       1
#define rcExists        2
#define rcNoExists      3
#define rcInvalid       4
#define rcBadHandle     5
#define rcBadArg        6
#define rcUnimplemented 7
#define rcOutOfMemory   8
#define rcNoPermission  9
#define rcBadVersion    10
#define rcDiskFull      11
#define rcInternal      12
#define rcNoFileHandles 13
#define rcFileChange    14
#define rcTooBig        15

/* following not from core engine: */
#define rcReadError     101

/* Errors for Error() */
#define wERRS_OOM               2   /* Out of memory */
#define wERRS_NOHELPPS          3   /* No help in print setup */
#define wERRS_NOHELPPR          4   /* No help while printing */
#define wERRS_FNF            1001   /* Cannot find file */
#define wERRS_NOTOPIC        1002   /* Topic does not exist */
#define wERRS_NOPRINTER      1003   /* Cannot print */
#define wERRS_PRINT          1004   /* Print error */
#define wERRS_EXPORT         1005   /* Can't copy to clipboard */
#define wERRS_BADFILE        1006   /* Not a Windows Help file */
#define wERRS_OLDFILE        1007   /* Cannot read help file */
#define wERRS_VIRUS          1011   /* Bad .EXE */
#define wERRS_BADDRIVE       1012   /* Invalid drive */
#define wERRS_WINCLASS       1014   /* Bad window class */
#define wERRS_BADKEYWORD     3012   /* Invalid keyword */
#define wERRS_BADPATHSPEC    3015   /* Invalid path */
#define wERRS_PATHNOTFOUND   3017   /* Path not found */
#define wERRS_DIALOGBOXOOM   3018   /* No memory for dialog */
#define wERRS_DISKFULL       5001   /* Disk is full */
#define wERRS_FSREADWRITE    5002   /* File I/O error */

/* Actions for LGetInfo() */
#define GI_NOTHING   0  /* Not used */
#define GI_INSTANCE  1  /* Application instance handle */
#define GI_MAINHWND  2  /* Main window handle */
#define GI_CURRHWND  3  /* Current window handle */
#define GI_HFS       4  /* Handle to file system in use */
#define GI_FGCOLOR   5  /* Foreground color used by app */
#define GI_BKCOLOR   6  /* Background color used by app */
#define GI_TOPICNO   7  /* Topic number */
#define GI_HPATH     8  /* Handle containing path */

/* Callback Function Table offsets: */
```

```
   #define HE_NotUsed           0
   #define HE_HfsOpenSz         1
   #define HE_RcCloseHfs        2
   #define HE_HfOpenHfs         3
   #define HE_RcCloseHf         4
   #define HE_LcbReadHf         5
   #define HE_LTellHf           6
   #define HE_LSeekHf           7
   #define HE_FEofHf            8
   #define HE_LcbSizeHf         9
   #define HE_FAccessHfs       10
   #define HE_RcLLInfoFromHf   11
   #define HE_RcLLInfoFromHfs  12
   #define HE_ErrorW           13
   #define HE_ErrorSz          14
   #define HE_GetInfo          15
   #define HE_API              16
```

Additional reference words: 3.10 3.51 4.00 LDLLHandler
KBCategory: kbtool
KBSubcategory: TlsHlp

Detecting Closure of Command Window from a Console App

PSS ID Number: Q102429

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

Win32 console applications run in a command window. For the console application to detect when the console is closing, register a console control handler and look for the following values in your case statement:

CTRL_CLOSE_EVENT	User closes the console
CTRL_LOGOFF_EVENT	User logs off
CTRL_SHUTDOWN_EVENT	User shuts down the system

For an example, see the CONSOLE sample. For more information, see the entry for SetConsoleCtrlhandler() in the Win32 application programming interface (API) reference.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseCon

Detecting Data on the Communications Port

PSS ID Number: Q118625

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

To detect whether there is any data available on the communications (COM) port without calling ReadFile(), use the ClearCommError() API. The field COMSTAT.CbInQue contains the number of bytes not yet read by a call to ReadFile().

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseCommapi

Detecting Keystrokes While a Menu Is Pulled Down

PSS ID Number: Q35930

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

In the Windows environment, there are two methods that an application can use to receive notification that a key is pressed while a menu is dropped down:

- The easier method is to process the WM_MENUCHAR message that is sent to an application when the user presses a key that does not correspond to any of the accelerator keys defined for the current menu.

- The other method is to use a message filter hook specified with the SetWindowsHook function. The hook function can process a message before it is dispatched to a dialog box, message box, or menu.

The hook functions and the WM_MENUCHAR message are documented in the Microsoft Windows SDK "Reference: Volume 1" for version 3.0 and in "Programmer's Reference, Volume 3: Messages, Structures, and Macros" for version 3.1.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrInp

Detecting Logoff from a Service

PSS ID Number: Q151424

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), for Windows NT, versions 3.5, 3.51, 4.0

SUMMARY
=======

Sometimes it is handy for a service to determine whether or not a user is logged on the system. For example, there are circumstances under which the service displays a dialog box and waits for a user to respond. If this is done when no user is logged on, the service is blocked until a user logs on.

Services can be notified when a user logs off a system but not when a user logs on. A service receives a notification from the system by installing a Console Control handler. The handler receives a CTRL_LOGOFF_EVENT when the interactive user logs off the system. For more information on Console Control handlers, please refer to the Win32 SDK Online Reference.

A service can determine whether an interactive user is logged on by determining whether the process specified in the following key is running:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
    \Winlogon
```

In this key, the process specified by the value shell is launched when an interactive user logs onto a system. If this process is running, it indicates that there is an interactive user logged on. To enumerate the processes on a Windows NT system, the service needs to examine the Windows NT performance counters. For more information, please refer to either the TLIST sample on the Win32 SDK or the Win32 SDK Online References.

Additional reference words: 3.50 3.51 4.00
KBCategory: kbprg
KBSubcategory: BseService

Detecting Logoff from a Service

PSS ID Number: Q104122

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

Sometimes it is handy for a service to know whether or not a user is logged on to the system. For example, suppose that there are circumstances under which the service displays a dialog box and waits for the user to respond. If this is done while the user is logged off, the service is blocked until the user logs on again.

Unfortunately, there is no direct way for a service to detect whether or not a user is logged on. There is, however, an indirect method. If you supply MB_DEFAULT_DESKTOP_ONLY as one of the flags in the fuStyle parameter of MessageBox(), the function will fail if no one is logged on or if a screen saver is running.

SetConsoleCtrlHandler can be used to install a function that will get called when the user logs off.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseService

Detecting the Presence of NetBIOS in Win32s

PSS ID Number: Q110844

The information in this article applies to:

- Microsoft Win32s versions 1.1 and 1.2

The way to determine whether NetBIOS is present from a Win32 application running on Win32s is to issue an invalid NetBIOS command (such as 0xB2) and check that the return code is 0x3 (Illegal command).

Additional reference words: 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Detecting Windows NT from an MS-DOS-Based Application

PSS ID Number: Q100290

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1 and 3.5

SUMMARY

There are calls that MS-DOS-based applications can make that are not supported under Windows NT. For example, calling Interrupt 25h to read the disk is not supported under Windows NT. Therefore, in some cases MS-DOS-based applications will need to know whether or not they are running under Windows NT.

Interrupt 21h, function 3306h can be used by MS-DOS-based applications to detect whether or not they are running under Windows NT. On return, registers BL and BH will contain the operating system major and minor numbers, respectively. If your application is running under Windows NT, the return will be:

```
BL = 5
BH = 50
```

MORE INFORMATION

Note that it is important to check both BL and BH, because MS-DOS 5.0 will also return a 5 in BL.

The following code demonstrates how to detect the operating system version from an MS-DOS-based application:

Sample Code

```
#include <stdio.h>
#include <stdlib.h>
#include <io.h>

void main()
{
   unsigned char cbh = 0;
   unsigned char cbl = 0;

   _asm {
      mov ax, 3306h
      int 21h
      mov cbh, bh
      mov cbl, bl
   }
   printf( "After int 21h\n" );
```

```
   printf( "%u, %u (bh, bl)\n", cbh, cbl );
}
```

Additional reference words: 3.10 3.50 determine
KBCategory: kbprg
KBSubcategory: SubSys

Detecting x86 Floating Point Coprocessor in Win32

PSS ID Number: Q124207

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

In Windows NT (x86) and Windows 95, floating point is emulated by the operating system, in the event that a numeric coprocessor is not present. This allows Win32-based applications to be compiled with floating point instructions present, which will be trapped by the operating system at runtime in the event that a coprocessor is not present. This behavior is transparent to the application, so it is difficult to detect.

In some cases, it is useful to execute code based on the presence of a numeric coprocessor, so this article explains how to do it.

MORE INFORMATION

One approach you can use to detect whether a coprocessor is present is to read the CR0 (System Control Register). This is not possible from Ring 3 application code under Windows NT, so a different approach is outlined below.

To determine whether a coprocessor is present on a computer using the x86 platform running Windows NT, you need to determine if the registry entry HKEY_LOCAL_MACHINE\\HARDWARE\\DESCRIPTION\\System\\FloatingPointProcessor is present. If this key is present, a numeric coprocessor is present.

On the MIPS and Alpha platforms, this registry key is not present because floating point support is built-in. The following function indicates whether a numeric coprocessor is present on Windows NT. If the function returns TRUE, a coprocessor is present. If the function returns FALSE, and GetLastError() indicates ERROR_RESOURCE_DATA_NOT_FOUND, a coprocessor is not present. Otherwise, an error occured while attempting to detect for a coprocessor. Some error checking is omitted, for brevity.

```
BOOL IsCoProcessorPresent(void)
{
    #define MY_ERROR_WRONG_OS 0x20000000
    HKEY hKey;
    SYSTEM_INFO SystemInfo;

    // return FALSE if we are not running under Windows NT
    // this should be expanded to cover alternative Win32 platforms
```

```
    if(!(GetVersion() & 0x7FFFFFFF))
    {
        SetLastError(MY_ERROR_WRONG_OS);
        return(FALSE);
    }

    // we return TRUE if we're not running on x86
    // other CPUs have built in floating-point, with no registry entry

    GetSystemInfo(&SystemInfo);

    if((SystemInfo.dwProcessorType != PROCESSOR_INTEL_386) &&
       (SystemInfo.dwProcessorType != PROCESSOR_INTEL_486) &&
       (SystemInfo.dwProcessorType != PROCESSOR_INTEL_PENTIUM))
    {
        return(TRUE);
    }

    if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,

"HARDWARE\\DESCRIPTION\\System\\FloatingPointProcessor",

                    0,
                    KEY_EXECUTE,
                    &hKey) != ERROR_SUCCESS)
    {
        // GetLastError() will indicate ERROR_RESOURCE_DATA_NOT_FOUND
        // if we can't find the key.  This indicates no coprocessor present
        return(FALSE);
    }

    RegCloseKey(hKey);
    return(TRUE);
}
```

Additional reference words: 3.10 3.50 4.00
KBCategory: kbprg kbcode
KBSubcategory: BseFltpt

Determining Available RGB Values of an Output Device

PSS ID Number: Q27225

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The best way to determine the colors supported by a device is to enumerate the solid pens from a device context (DC) associated with that device. The EnumObjects function enumerates the pens supported by a specified device and calls a callback function for each pen. To do so, EnumObjects requires three parameters: a device context associated with the desired device as the hDC parameter, OBJ_PEN as the value of the nOjbectType parameter, and the procedure-instance address of a callback function as the lpObjectFunc parameter.

MORE INFORMATION
================

The first parameter for the callback function, lpLogObject, points to a LOGPEN data structure that describes each enumerated pen. When the lopnStyle field of the LOGPEN structure contains the PS_SOLID value, the application can retrieve and store the value of the lopnColor field.

NOTE: For true color devices (devices that support more than 8 bits-per-pixel of color resolution), Windows enumerates only a subset of the supported pens. These devices support almost any color.

The following code demonstrates the process described above:

```
int cMaxRGB, nCnt, nSolid; // Global variables for system RGB colors

void GetColorList(HWND hWnd)
{
   HDC hdc;
   FARPROC lpProcCallback;
   HANDLE hmem;
   BYTE FAR *lpmem;

   nCnt = nSolid = 0;
   hdc = GetDC(hWnd);
   cMaxRGB = GetDeviceCaps(hdc, NUMCOLORS);
   if (cMaxRGB >= 0x7FFF)
```

```
      return; // All colors available. Enumeration not required.

   lpProcCallback = MakeProcInstance(Callback, hInst);

   // Allocate space for color table
   hmem = GlobalAlloc(GHND, sizeof(COLORREF) * cMaxRGB);
   lpmem = GlobalLock(hmem);

   EnumObjects(hdc, OBJ_PEN, lpProcCallback, lpmem);

   FreeProcInstance(lpProcCallback);

   // Use the color table stored in the first nSolid entries of a
   // COLORREF array stored in lpmem.

   GlobalUnlock(hmem);
   GlobalFree(hmem);
   ReleaseDC(hWnd, hdc);
   return;
}
```

The source for the enumeration callback function is below. The callback function must be listed in the EXPORTS section of the module definition (DEF) file.

```
int FAR PASCAL Callback(LPLOGPEN lpLogPen, LPSTR lpData)
{
   nCnt++;
   if (lpLogPen->lopnStyle == PS_SOLID) // solid pen
      {
      COLORREF FAR *lpDest = (COLORREF FAR *)lpData + nSolid++;

      *lpDest = lpLogPen->lopnColor; // remember the solid color
      }

   if (nCnt >= cMaxRGB)
      return 0;
   return 1;
}
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiPal

Determining Connected Joysticks Under Windows 95

PSS ID Number: Q133065

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK)

SUMMARY
=======

The joystick API joyGetNumDevs() behaves differently on Windows 95 than it does in other implementations, so a different technique, involving joyGetPosEx(), is required to determine the number of joysticks available during game play.

This article provides the code for a function called JoysticksConnected() that demonstrates how to determine the number of connected joysticks.

MORE INFORMATION
================

Under Windows 95, the joyGetNumDevs() API returns: 0 if no joystick drivers have been installed or 16 (the highest possible number of connected joysticks) in all other cases. This behavior is by design. To determine the actual number of joysticks installed, use the joyGetPosEx() API.

The design for Windows 95 joystick drivers calls for the driver to support two independent joystick input devices. Therefore, during driver installation, two registry keys are created, and initialized. The keys correspond to joystick 0 and joystick 1, respectively.

If no joystick support is available (no drivers), joyGetNumDevs() returns 0.

When joyGetNumDevs() returns a non-zero value, joyGetPosEx() returns JOYERR_NOERRROR or JOYERR_PARMS (defined in the file MMSYSTEM.H). You can examine the return values and determine the number of joysticks connected as shown below. NOTE: when linking, WINMM.LIB must be from the Win32 SDK:

```
   int JoysticksConnected( )
   {
   // determine number of joysticks installed in Windows 95

      JOYINFOEX info;      // extended information

      int njoyId = 0;      // first joystick

      int nConnected = 0;  // goal - number of joysticks connected

      MMRESULT dwResult;   // examine return values

   // Loop through all possible joystick IDs until we get the error
   // JOYERR_PARMS. Count the number of times we get JOYERR_NOERROR
```

```
   // indicating an installed joystick driver with a joystick currently
   // attached to the port.

      while ((dwResult = joyGetPosEx(njoyId++,&info)) != JOYERR_PARMS)

      if (dwResult == JOYERR_NOERROR)
         ++nConnected;    // the count of connected joysticks

      return nConnected;  // return the count of joysticks found

   } // JoysticksConnected
```

Additional reference words: 4.00 JOYERR connect kbinf
KBCategory: kbmm
KBSubcategory: MMJoy

Determining Selected Items in a Multiselection List Box

PSS ID Number: Q71759

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

To obtain the indexes of all the selected items in a multiselection list box, the LB_GETSELITEMS message should be sent to the list box.

The message LB_GETCURSEL cannot be used for this purpose because it is designed for use in single-selection list boxes.

Another approach is to send one LB_GETSEL message for every item of the multiselection list box to get its selection state. If the item is selected, LB_GETSEL returns a positive number. The indexes can be built into an array of selected items.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Determining System Version from a Win32-based Application

PSS ID Number: Q92395

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s versions 1.2, 1.25a, and 1.3

SUMMARY

In order to create a Win32-based application that takes advantage of the features of each platform, it is necessary to determine the operating system on which the application is currently running.

You can use GetVersion() or GetVersionEx() to determine what operating system and version your application is running under. NOTE: GetVersion() is supported on Windows 3.1, but GetVersionEx() is new to the Win32 API. A Win32-based application might be running under MS-DOS/Windows using the Win32s extension, Windows NT Workstation, Windows NT Server, or Windows 95.

MORE INFORMATION

According to the documentation, the return value of GetVersion() is a DWORD that specifies the major and minor version numbers. GetVersionEx() uses members of the OSVERSIONINFO structure (dwMajorVersion and dwMinorVersion).

The following table shows the return values from GetVersion() under various environments:

Environment	LOWORD	HIWORD
Win32s on Windows 3.1	Windows version 3.1	RESERVED *
Windows NT	Windows version	RESERVED **
Windows 95	Windows version 4.0	RESERVED ***

  * The highest bit is 1. The remaining bits specify build number. Note that the version of MS-DOS cannot be determined as it can under Windows 3.x.
 ** The highest bit is 0. The remaining bits specify build number.
*** The highest bit is 1. The remaining bits are reserved.

The following sample code can be used to test the values returned by GetVersion().

Sample Code 1

```
#include <windows.h>

void main()
{

   DWORD dwVersion;
   char szVersion[80];

   dwVersion = GetVersion();

   if (dwVersion < 0x80000000) {
   // Windows NT
       wsprintf (szVersion, "Microsoft Windows NT %u.%u (Build: %u)",
                 (DWORD)(LOBYTE(LOWORD(dwVersion))),
                 (DWORD)(HIBYTE(LOWORD(dwVersion))),
                 (DWORD)(HIWORD(dwVersion)));
   }
   else if (LOBYTE(LOWORD(dwVersion))<4) {
   // Win32s
       wsprintf (szVersion, "Microsoft Win32s %u.%u (Build: %u)",
                 (DWORD)(LOBYTE(LOWORD(dwVersion))),
                 (DWORD)(HIBYTE(LOWORD(dwVersion))),
                 (DWORD)(HIWORD(dwVersion) & ~0x8000));
   } else {
   // Windows 95
       wsprintf (szVersion, "Microsoft Windows 95 %u.%u (Build: %u)",
                 (DWORD)(LOBYTE(LOWORD(dwVersion))),
                 (DWORD)(HIBYTE(LOWORD(dwVersion))),
                 (DWORD)(HIWORD(dwVersion) & ~0x8000));
   }

   MessageBox( NULL, szVersion, "Version Check", MB_OK );
}
```

The following sample code can be used to test the values returned by GetVersionEx(). NOTE: The actual build number is derived by masking dwBuildNumber with 0xFFFF.

Sample Code 2

```
{
   OSVERSIONINFO osvi;
   char  szVersion [80];

   memset(&osvi, 0, sizeof(OSVERSIONINFO));
   osvi.dwOSVersionInfoSize = sizeof (OSVERSIONINFO);
   GetVersionEx (&osvi);
```

```
   if (osvi.dwPlatformId == VER_PLATFORM_WIN32s)
      wsprintf (szVersion, "Microsoft Win32s %d.%d (Build %d)",
                osvi.dwMajorVersion,
                osvi.dwMinorVersion,
                osvi.dwBuildNumber & 0xFFFF);

   else if (osvi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS)
      wsprintf (szVersion, "Microsoft Windows 95 %d.%d (Build %d)",
                osvi.dwMajorVersion,
                osvi.dwMinorVersion,
                osvi.dwBuildNumber & 0xFFFF);

   else if (osvi.dwPlatformId == VER_PLATFORM_WIN32_NT)
      wsprintf (szVersion, "Microsoft Windows NT %d.%d (Build %d)",
                osvi.dwMajorVersion,
                osvi.dwMinorVersion,
                osvi.dwBuildNumber & 0xFFFF);

   MessageBox( NULL, szVersion, "Version Check", MB_OK );
}
```

In order to distinguish between Windows NT Workstation and Windows NT Server, use the registry API to query the following:

```
   \HKEY_LOCAL_MACHINE\SYSTEM
   \CurrentControlSet
   \Control
   \ProductOptions
```

The result will be one of the following:

```
   WINNT      Windows NT Workstation is running.
   SERVERNT   Windows NT Server (3.5 or later) is running.
   LANMANNT   Windows NT Advanced Server (3.1) is running.
```

Additional reference words: 1.20 1.30 3.10 3.50 4.00 detect
KBCategory: kbprg
KBSubcategory: BseMisc

Determining the Maximum Allowed Access for an Object

PSS ID Number: Q115945

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

The AccessCheck() API call can be used to determine the maximum access to an object allowed for a subject. (In this article, a subject means a program running in a specific user's security context.) When using AccessCheck() for this purpose, perform the following steps:

1. Obtain a security descriptor that has owner, group, and DACL information.

2. If you are not impersonating a client, obtain an impersonation token by calling ImpersonateSelf. This token is passed as the client token in the AccessCheck() call.

3. Create a generic mapping structure. The contents of this structure will vary depending on the object being used.

4. Call AccessCheck() and request "MAXIMUM_ALLOWED" as the desired access.

If the AccessCheck() call succeeds after the above steps have been completed, the dwGrantedAccess parameter to AccessCheck() contains a mask of the object-specific rights that are granted by the security descriptor.

MORE INFORMATION

In most situations, you should not use this method of access determination. If you need access to an object to perform a task, simply try to open the object using the required access.

The AccessCheck() API is mainly intended for use with private objects created by an application. However, it can be used with predefined objects. The generic mapping values and specific rights for many of the predefined objects (files and so forth) may be found in WINNT.H.

REFERENCES

Please see the Security Overview in the "Win32 Programmer's Reference" and the "Win32 SDK API Reference" for more information.

Additional reference words: 3.10 3.50

KBCategory: kbprg
KBSubcategory: BseSecurity

Determining the Network Protocol Used By Named Pipes

PSS ID Number: Q126766

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.5

This article discusses how to determine or set the network protocol used by named pipes.

Named pipes are implemented using the server message block (SMB) redirector and server. As such, they can use whatever protocols are bound into the server and the client.

Both the redirector and the server maintain independent lists of transports that they are active on. The redirector contacts the remote server. The redirector will use the highest priority transport that both the client and the server support.

The priority for the transports is set using the Network control panel applet. Go into the Bindings button, select Workstation, and use the up and down buttons to rearrange the order that the redirector will use while trying to connect to the remote server.

Additional reference words: 3.50
KBCategory: kbprg
KBSubcategory: BseIpc

Determining the Number of Visible Items in a List Box

PSS ID Number: Q78952

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

To determine the number of items that are currently visible in a list box, an application must consider the following cases:

1. There are many items in the list box (some items are not visible).

2. There are few items in the list box (the bottom area of the list box is empty).

3. The heights of the items may vary (an owner-draw list box or use of a font other than the system default).

MORE INFORMATION
================

The following code segment can be used to determine the number of items visible in a list box:

Sample Code

```
int ntop, nCount, nRectheight, nVisibleItems;
RECT rc;

ntop = SendMessage(hwndList, LB_GETTOPINDEX, 0, 0);
                                // Top item index.

nCount = SendMessage(hwndList, LB_GETCOUNT, 0, 0);
                                // Number of total items.

GetClientRect(hwndList, &rc);
                                // Get list box rectangle.

nRectheight = rc.bottom - rc.top;
                                // Compute list box height.

nVisibleItems = 0;              // Initialize counter.

while ((nRectheight > 0) && (ntop < nCount))
```

```
                        // Loop until the bottom of the list box
                        // or the last item has been reached.
   {
   SendMessage(hwndList, LB_GETITEMRECT, ntop, (DWORD)(&itemrect));
                               // Get current line's rectangle.

   nRectheight = nRectheight - (itemrect.bottom - itemrect.top);
                               // Subtract current line height.

   nVisibleItems++;            // Increase item count.
   ntop++;                     // Move to the next line.
   }
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Determining the Topmost Pop-Up Window

PSS ID Number: Q66943

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

When an application has many pop-up child windows (with a common parent window), the GetNextWindow() function can be used when one pop-up window is closed to determine the next topmost pop-up window that remains.

The following code fragment shows a window procedure for simple pop-up windows (modified from the PARTY program in Petzold's "Programming Windows"). In the WM_CLOSE case, the handle received by the pop-up window procedure is the handle of the pop-up to be closed. This sample activates the topmost pop-up window that remains by giving it the focus.

```
long FAR PASCAL PopupWndProc (hWnd, iMessage, wParam, lParam)
     HWND     hWnd;
     unsigned iMessage;
     WORD     wParam;
     LONG     lParam;
     {
     HWND     hWndPopup;

     switch (iMessage)
        {
        case WM_CLOSE:
            hWndPopup = GetNextWindow(hWnd, GW_HWNDNEXT);
            if (hWndPopup)
                SetFocus(hWndPopup);
            break;
        }

     return DefWindowProc (hWnd, iMessage, wParam, lParam) ;
     }
```

NOTE: In Windows 3.1, two messages are sent to an application when its Z-order is changing: WM_WINDOWPOSCHANGING and WM_WINDOWPOSCHANGED. When a window is closed (as in the example shown above) these two message will be sent to all window procedures.

For additional information on changing the Z-order of an MDI window, query on the following words in the Microsoft Knowledge Base:

   WM_WINDOWPOSCHANGED and MDICREATESTRUCT and WS_EX_TOPMOST

Additional reference word: 3.00 3.10 3.50 3.51 4.00 95

KBCategory: kbui
KBSubcategory: UsrWndw

Determining the Visible Area of a Multiline Edit Control

PSS ID Number: Q88387

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The multiline edit control provided with Microsoft Windows versions 3.0 and 3.1 does not provide a mechanism that allows an application to determine the currently visible lines of text. This article outlines an algorithm to provide that functionality.

MORE INFORMATION

The general idea is to determine the first and last visible lines and obtain the text of those lines from the edit control. The following steps detail this process:

1. A Windows-3.1-based application can use the newly available message, EM_GETFIRSTVISIBLELINE, to determine the topmost visible line.

   A Windows-3.0-based application can use a technique described in the following Microsoft Knowledge Base article to determine the line number of the first visible line:

   ARTICLE-ID: Q68572
   TITLE : Caret Position & Line Numbers in Multiline Edit Controls

2. Obtain the edit control's formatting rectangle using EM_GETRECT. Determine the rectangle's height using this formula:

   ```
   nFmtRectHeight = rect.bottom - rect.top;
   ```

3. Obtain the line spacing of the font used by the edit control to display the text. Use the WM_GETFONT message to determine the font used by the edit control. Select this font into a display context and use the GetTextMetrics function. The tmHeight field of the resulting TEXTMETRIC structure is the line spacing.

4. Divide the formatting rectangle's height (step 2) with the line spacing (step 3) to determine the number of lines. Compute the line number of the last visible line based on the first visible line (step 1) and the number of visible lines.

5. Use EM_GETLINE for each line number from the first visible line to the last visible line to determine the visible lines of text. Remember that the last visible line may not necessarily be at the bottom of the edit control (the control may only be half full). To detect this case, use EM_GETLINECOUNT to know the last line and compare its number with the last visible line. If the last line number is less than the last visible line, your application should use EM_GETLINE only on lines between the first and the last line.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Determining Visible Window Area When Windows Overlap

PSS ID Number: Q75236

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

There is no Windows API that reports the portion of an application's window that is not obscured by other windows. To determine which areas of the window are covered, it is necessary to walk through the window list managed by Windows.

Each window that precedes the application's window is "above" that window on the screen. Using the IntersectRect() function, check the rectangle of the window with any windows above to see if they intersect. Any window that is above the application's window and intersects its window rectangle obscures part of the application's window. By accumulating the positions of all windows that overlap the application's window, it is possible to determine which areas of the window are covered and which are not.

MORE INFORMATION

The following sample code demonstrates this procedure:

```
  GetWindowRect(hWnd, &rMyRect);    /*  Get the window dimensions
                                     *  for the current window.
                                     */
      /*  Start from the current window and use the GetWindow()
       *  function to move through the previous window handles.
       */
  for (hPrevWnd = hWnd;
      (hNextWnd = GetWindow(hPrevWnd, GW_HWNDPREV)) != NULL;
       hPrevWnd = hNextWnd)
    {
      /*  Get the window rectangle dimensions of the window that
       *  is higher Z-Order than the application's window.
       */
    GetWindowRect(hNextWnd, &rOtherRect);

      /*  Check to see if this window is visible and if intersects
       *  with the rectangle of the application's window. If it does,
       *  call MessageBeep(). This intersection is an area of this
       *  application's window that is not visible.
```

```
       */
    if (IsWindowVisible(hNextWnd) &&
            IntersectRect(&rDestRect, &rMyRect, &rOtherRect))
      {
      MessageBeep(0);
      }
    }
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Determining Whether a WOW App is Running in Enhanced Mode

PSS ID Number: Q101893

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY

Windows NT and Windows NT Advanced Server will run a Windows 3.1 application in 386 enhanced mode on X86 machines (standard mode on RISC machines). The proper way to determine whether a Windows 3.1 application is in enhanced mode is to call GetWinFlags() and do a bit test for WF_ENHANCED. This method is described on pages 486-487 in the Windows 3.1 Software Development Kit (SDK) "Programmer's Reference, Volume 2: Functions."

MORE INFORMATION

Calling Interrupt 2F with AX=1600h

This method, which is described in the Windows 3.1 Device Development Kit (DDK) checks to see whether a 386 memory manager is running. If Windows 3.1 is running in enhanced mode, it returns AL = 3 and 2 for standard mode. Windows NT's WOW (Windows 16 on Windows NT) returns AL=0, which means enhanced mode Windows is not running.

DWORD GetWinFlags()

The GetWinFlags() function retrieves the current Windows system and memory configuration.

The configuration returned by GetWinFlags() can be a combination of the following values:

Value	Meaning
WF_80x87	System contains an Intel math coprocessor.
WF_CPU286	System CPU is an 80286.
WF_CPU386	System CPU is an 80386.
WF_CPU486	System CPU is an i486.
WF_ENHANCED	Windows is running in 386-enhanced mode. The WF_PMODE flag is always set when WF_ENHANCED is set.
WF_PAGING	Windows is running on a system with paged memory.
WF_PMODE	Windows is running in protected mode. In Windows 3.1, this flag is always set.
WF_STANDARD	Windows is running in standard mode. The WF_PMODE flag is always set when WF_STANDARD is set.

```
   WF_WIN286      Same as WF_STANDARD.
   WF_WIN386      Same as WF_ENHANCED.
```

NOTE: When running in Windows NT, WF_WINNT will also be returned to tell the 16-bit Windows-based application that you are running in Windows NT.

Example:

The following example uses the GetWinFlags() function to display information about the current Windows system configuration:

Sample Code

```
int len;
char szBuf[80];
DWORD dwFlags;

dwFlags = GetWinFlags();

len = sprintf(szBuf, "system %s a coprocessor",
    (dwFlags & WF_80x87) ? "contains" : "does not contain");
TextOut(hdc, 10, 15, szBuf, len);

len = sprintf(szBuf, "processor is an %s",
    (dwFlags & WF_CPU286) ? "80286" :
    (dwFlags & WF_CPU386) ? "80386" :
    (dwFlags & WF_CPU486) ? "i486" : "unknown");
TextOut(hdc, 10, 30, szBuf, len);

len = sprintf(szBuf, "running in %s mode",
    (dwFlags & WF_ENHANCED) ? "enhanced" : "standard");
TextOut(hdc, 10, 45, szBuf, len);

len = sprintf(szBuf, "%s WLO",
    (dwFlags & WF_WLO) ? "using" : "not using");
TextOut(hdc, 10, 60, szBuf, len);
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: SubSys

Determining Whether an Application is Console or GUI

PSS ID Number: Q90493

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

=======

In order to determine whether an application is console or GUI, you must parse the EXEheader. The header contains a field called 'Subsystem'. This field determines both the subsystem the application is to run under and the type of interface it requires. The values consist of:

IMAGE_SUBSYSTEM_NATIVE	1
IMAGE_SUBSYSTEM_WINDOWS_GUI	2
IMAGE_SUBSYSTEM_WINDOWS_CUI	3
IMAGE_SUBSYSTEM_OS2_CUI	5
IMAGE_SUBSYSTEM_POSIX_CUI	7

MORE INFORMATION

================

Sample Code

```
#include <windows.h>
#include <winnt.h>

VOID  main(int, char **);
DWORD AbsoluteSeek(HANDLE, DWORD);
VOID  ReadBytes(HANDLE, LPVOID, DWORD);
VOID  WriteBytes(HANDLE, LPVOID, DWORD);
VOID  CopySection(HANDLE, HANDLE, DWORD);

#define XFER_BUFFER_SIZE 2048

VOID
main(int argc, char *argv[])
{
    HANDLE hImage;

    DWORD  bytes;
    DWORD  iSection;
    DWORD  SectionOffset;
    DWORD  CoffHeaderOffset;
    DWORD  MoreDosHeader[16];

    ULONG  ntSignature;
```

```
IMAGE_DOS_HEADER      image_dos_header;
IMAGE_FILE_HEADER     image_file_header;
IMAGE_OPTIONAL_HEADER image_optional_header;
IMAGE_SECTION_HEADER  image_section_header;

if (argc != 2)
{
    printf("USAGE: %s program_file_name\n", argv[1]);
    exit(1);
}

/*
 *  Open the reference file.
 */
hImage = CreateFile(argv[1],
                    GENERIC_READ,
                    FILE_SHARE_READ,
                    NULL,
                    OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);

if (INVALID_HANDLE_VALUE == hImage)
{
    printf("Could not open %s, error %lu\n", argv[1], GetLastError());
    exit(1);
}

/*
 *  Read the MS-DOS image header.
 */
ReadBytes(hImage,
          &image_dos_header,
          sizeof(IMAGE_DOS_HEADER));

if (IMAGE_DOS_SIGNATURE != image_dos_header.e_magic)
{
    printf("Sorry, I do not understand this file.\n");
    exit(1);
}

/*
 *  Read more MS-DOS header.       */
ReadBytes(hImage,
          MoreDosHeader,
          sizeof(MoreDosHeader));

/*
 *  Get actual COFF header.
 */
CoffHeaderOffset = AbsoluteSeek(hImage, image_dos_header.e_lfanew) +
                   sizeof(ULONG);

ReadBytes (hImage, &ntSignature, sizeof(ULONG));
```

```
    if (IMAGE_NT_SIGNATURE != ntSignature)
    {
     printf("Missing NT signature. Unknown file type.\n");
     exit(1);
    }

    SectionOffset = CoffHeaderOffset + IMAGE_SIZEOF_FILE_HEADER +
                    IMAGE_SIZEOF_NT_OPTIONAL_HEADER;

    ReadBytes(hImage,
              &image_file_header,
              IMAGE_SIZEOF_FILE_HEADER);

    /*
     *  Read optional header.
     */
    ReadBytes(hImage,
              &image_optional_header,
              IMAGE_SIZEOF_NT_OPTIONAL_HEADER);

    switch (image_optional_header.Subsystem)
    {
    case IMAGE_SUBSYSTEM_UNKNOWN:
        printf("Type is unknown.\n");
        break;

    case IMAGE_SUBSYSTEM_NATIVE:
        printf("Type is native.\n");
        break;

    case IMAGE_SUBSYSTEM_WINDOWS_GUI:
        printf("Type is Windows GUI.\n");
        break;

    case IMAGE_SUBSYSTEM_WINDOWS_CUI:
        printf("Type is Windows CUI.\n");
        break;

    case IMAGE_SUBSYSTEM_OS2_CUI:
        printf("Type is OS/2 CUI.\n");
        break;

    case IMAGE_SUBSYSTEM_POSIX_CUI:
        printf("Type is POSIX CUI.\n");
        break;

    default:
        printf("Unknown type %u.\n", image_optional_header.Subsystem);
        break;
    }
}

DWORD
AbsoluteSeek(HANDLE hFile,
```

```
            DWORD  offset)
{
    DWORD newOffset;

    if ((newOffset = SetFilePointer(hFile,
                                    offset,
                                    NULL,
                                    FILE_BEGIN)) == 0xFFFFFFFF)
    {
        printf("SetFilePointer failed, error %lu.\n", GetLastError());
        exit(1);
    }

    return newOffset;
}

VOID
ReadBytes(HANDLE hFile,
          LPVOID buffer,
          DWORD  size)
{
    DWORD bytes;

    if (!ReadFile(hFile,
                  buffer,
                  size,
                  &bytes,
                  NULL))
    {
        printf("ReadFile failed, error %lu.\n", GetLastError());
        exit(1);
    }
    else if (size != bytes)
    {
        printf("Read the wrong number of bytes, expected %lu, got %lu.\n",
               size, bytes);
        exit(1);
    }
}
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

Determining Whether the User is an Administrator

PSS ID Number: Q118626

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

To determine whether or not a user is an administrator, you need to examine the user's access token with GetTokenInformation(). The access token represents the user's privileges and the groups to which the user belongs.

When a user starts an application, the access token associated with that process is the user's access token. To get the process token (and therefore the user's token), use OpenProcessToken().

The sample code below uses the APIs mentioned in the previous paragraph to test whether or not the current user is an administrator on the local machine:

Sample code

```
   /* BOOL IsAdmin(void)

      returns TRUE if user is an admin
              FALSE if user is not an admin
   */

   BOOL IsAdmin(void)
   {
      HANDLE hAccessToken;
      UCHAR InfoBuffer[1024];
      PTOKEN_GROUPS ptgGroups = (PTOKEN_GROUPS)InfoBuffer;
      DWORD dwInfoBufferSize;
      PSID psidAdministrators;
      SID_IDENTIFIER_AUTHORITY siaNtAuthority = SECURITY_NT_AUTHORITY;
      UINT x;
      BOOL bSuccess;

      if(!OpenProcessToken(GetCurrentProcess(),TOKEN_READ,&hAccessToken))
         return(FALSE);

      bSuccess = GetTokenInformation(hAccessToken,TokenGroups,InfoBuffer,
         1024, &dwInfoBufferSize);

      CloseHandle(hAccessToken);

      if( !bSucess )
         return FALSE;

      if(!AllocateAndInitializeSid(&siaNtAuthority, 2,
```

```
         SECURITY_BUILTIN_DOMAIN_RID,
         DOMAIN_ALIAS_RID_ADMINS,
         0, 0, 0, 0, 0, 0,
         &psidAdministrators))
         return FALSE;

   // assume that we don't find the admin SID.
      bSuccess = FALSE;

      for(x=0;x<ptgGroups->GroupCount;x++)
      {
         if( EqualSid(psidAdministrators, ptgGroups->Groups[x].Sid) )
         {
            bSuccess = TRUE;
            break;
         }

      }
      FreeSid(psidAdministrators);
      return bSuccess;
   }
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Developing Win32-Based GDI Apps for Windows 95 and Windows NT

PSS ID Number: Q136989

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0

SUMMARY

=======

When developing a Win32 application, you need to take certain programming aspects into consideration if the application is expected to be run under both Windows 95 and Windows NT.

For example, Windows 95, like Windows 3.x, still uses a 16-bit world coordinate system, while Windows NT uses a 32-bit coordinate system. In addition, Windows 95 does not support the GM_ADVANCED graphics mode option that Windows NT supports. The GetVersionEx() function can be used to determine the platform under which your application is running.

MORE INFORMATION

================

Because the SetGraphicsMode() mode function only accepts GM_COMPATIBLE in Windows 95, functions like SetWorldTransform() or ModifyWorldTransform() cannot be called to set or modify the world transformation for the specified device context. Windows NT supports this functionality. However, Windows 95 ensures that enhanced metafiles look the same in Windows 95 as they do in Windows NT. This means that enhanced metafiles are assumed to be played in GM_ADVANCED mode. If you want your GDI graphical output to look the same in both Windows 95 and Windows NT, it is recommended that you use GM_COMPATIBLE on both platforms. If you use GM_ADVANCED in Windows NT and GM_COMPATIBLE in Windows 95, for example, your rectangles would look different because a GM_COMPATIBLE system excludes bottom and rightmost edges when it draws rectangles, while a GM_ADVANCED system includes them. Also, the way rotated TrueType fonts are drawn differs.

If you pass 32-bit coordinates to GDI functions in Windows 95, the system truncates the upper 16 bits of the coordinates before actually performing the function. If you want to run under both Windows NT and Windows 95, then you need to keep this in mind. When using the Chord(), Pie(), Arc(), and RoundRect() functions under Windows 95, the sum of the coordinates sent to the functions cannot exceed 32K. Also, the sum of the left and right or top and bottom fields can not exceed 32K.

No matter what platform you are running, whether it is Windows 95 or Windows NT, a win32-based application must still delete all GDI objects when they're not needed. Logical objects should be created and deleted as they are needed. This will ensure that adequate space is always available for logical GDI objects because Windows 95 places them in the local 16-bit GDI heap, which is limited to 64K. All regions should be deleted with DeleteObject(). In addition, all bitmaps, brushes, extended pens, fonts, regular pens, and palettes should be deleted with DeleteObject(). A

metafile DC should be deleted with CloseMetaFile(), and a metafile should be deleted with DeleteMetaFile(). An enhanced metafile DC should be deleted with CloseEnhMetaFile() and an enhanced metafile should be deleted with DeleteEnhMetaFile(). A memory DC created with CreateCompatibleDC() should always be deleted with the DeleteDC() function, a DC created with CreateDC() should be deleted with DeleteDC(), and a DC obtained by GetDC() should be released with ReleaseDC(). Physical GDI objects exist in global memory, so they are not limited to the GDI 16-bit local heap.

Both Windows 95 and Windows NT free all GDI resources owned by a 32-bit process when that process terminates. Windows 95 also cleans up any GDI resources of 16-bit processes if it is marked as a 4.0 application. Windows 95 doesn't immediately clean up GDI resources for 16-bit applications marked with a version less than 4.0. The system waits until all 16-bit applications have finished running. Then all GDI resources allocated by previous 16-bit applications are cleaned up. In Windows 95, regions are allocated from the 32-bit heap, so they can be as large as available memory. This is a great step forward from Windows 3.x where regions were limited to 64K. However, the number of region handles cannot exceed 16K.

In Windows 95, if you try to delete a GDI drawing object while it is selected into a DC with DeleteObject(), the call to DeleteObject() succeeds, but the result of the call is a non-functioning GDI object. In Windows NT, a call to DeleteObject() on a GDI object selected into a DC will fail. You should always clear the selection of a GDI object before calling DeleteObject() on it.

The Windows NT operating system automatically tracks the origin of all window-managed DC's and adjusts their brushes to maintain an alignment of patterns on the surface. Windows 95 doesn't support automatic tracking of the brush origin, so an application must call the following three functions to align a brush each time it paints using a patterned brush:

```
UnrealizeObject()
SetBrushOrgEx()
SelectObject()
```

When creating patterned brushes with either CreatePatternBrush() or CreateDIBPatternBrush(), Windows 95 doesn't support bitmaps or DIBs greater than 8 by 8 pixels. Windows NT does not have this limitation. When writing an application that must be run under both platforms, you need to use a bitmap or DIB that is 8 by 8 pixels or less.

Windows NT provides a wide variety of pens. There are some limitations on pens in Windows 95:

- The ExtCreatePen() function only supports the PS_SOLID style for geometric lines.
- The PS_USERSTYLE and PS_ALTERNATE styles are not supported.
- Dashed or dotted pen styles like PS_DASH, for example, are not supported in wide lines.

The following pen styles for geometric lines are supported in paths only in Windows 95:

PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT, PS_JOIN_BEVEL, PS_JOIN_MITER, and PS_JOIN_ROUND

The ExtCreateRegion() function fails in Windows 95 if the transformation matrix is anything other than a scaling or translation of the region. This is because world transforms that involve either shearing or rotations are not supported in Windows 95.

The return value to GetDIBits() is also different in Windows 95 from Windows NT when the "lpvBits" parameter is NULL. In Windows 95, if the lpvBits parameter is NULL and GetDIBits successfully fills the BITMAPINFO structure, the return value is the total number of scan lines in the bitmap. In Windows NT, If the lpvBits parameter is NULL and GetDIBits successfully fills the BITMAPINFO structure, the return value is non-zero.

Windows 95 only records the following functions in a path:

ExtTextOut	LineTo	MoveToEx	PolyBezier
PolyBezierTo	Polygon	Polyline	PolylineTo
PolyPolygon	PolyPolyline	TextOut	

To find a complete list of functions supported in Windows 95 and Windows NT, view the WIN32API.CSV file in the Lib directory of the Microsoft Visual C++ compiler or Win32 SDK.

REFERENCES

For more information, please see:

- The Win32api.csv file in the Lib directory of the Microsoft Visual C++ compiler or Win32 SDK.

- "Programmer's Guide to Microsoft Windows 95," Microsoft Press.

Additional reference words: kbinf 4.00
KBCategory: kbgraphic
KBSubcategory: GdiMisc

Development Tools Do Not Accept Unicode Text

PSS ID Number: Q106065

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1 and 3.5

SUMMARY
=======

Neither the Win32 SDK tools or Visual C++ (VC++) 32-bit Edition tools support Unicode text. In fact, the C/C++ Language specification says that the source files are to be written in 7-bit ANSI.

MORE INFORMATION
================

For example, language-specific resources cannot be specified in Unicode in the .RC file because RC does not accept the Unicode text. Although the message compiler has flags for Unicode, the flags are not implemented.

To convert to and from Unicode text, use the UCONVERT utility included in your MSTOOLS\BIN directory. The source for UCONVERT in the 3.1 SDK is in MSTOOLS\SAMPLES\SDKTOOLS\UCONVERT. The source for UCONVERT in the 3.5 SDK is in MSTOOLS\SAMPLES\SDKTOOLS\WINNT\UCONVERT.

The long term solution that Microsoft is working on are Resource Localization Tools and other methods that will allow the user to localize the strings in a GUI editor, running on the target machine.

Note that it is possible to specify Unicode escapes in L-quoted strings. The following is quoted from "Common Statement Parameters" in RC.HLP:

> By default, the characters listed between the double quotation marks are ANSI characters and escape sequences are interpreted as byte escape sequences. If the string is preceded by the L prefix, the string is a wide-character string and escape sequences are interpreted as two-byte escape sequences that specify Unicode characters. If a double quotation mark is required in the text, you must include the double quotation mark twice or use the \" escape sequence.

Another alternative is to use user-defined resources and include a binary (Unicode) file.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: WIntlDev

DEVMODE and dmSpecVersion

PSS ID Number: Q96282

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The dmSpecVersion field of the DEVMODE structure is intended for printer driver use only; no application programs should test this field. The purpose of this field is for new printer drivers to be able to recognize and handle DEVMODE structures created according to previous DEVMODE structure specification.

MORE INFORMATION
================

The DEVMODE structure is used for printer and (occasionally) display drivers when initializing. This structure is tied to the driver--not the operating system. The dmSpecVersion field does not allow an application to determine which platform (Windows version 3.1, Windows on Windows, Win32) the application is running in.

When an application fills a DEVMODE structure, it should set the dmSpecVersion field to DM_SPECVERSION. This identifies the version of the DEVMODE structure the application is generating.

If the application is querying to understand an unknown device, then special attention should be paid to the dmFields, dmSize, and dmDriverExtra fields. These fields are a reliable means of understanding what fields in the DEVMODE structure are readable.

The DEVMODE structure consists of public and private parts. The dmSpecVersion field applies to the public part. Any previously defined fields are not altered when the DEVMODE specification is updated--more fields are merely added to the end of the structure. This can mean fields used in the previous specification are ignored in a later specification. This functionality is managed by one bitfield describing what fields a driver actually uses. The new drivers just switch off the obsolete fields.

Applications using DEVMODE should always use the dmSize and dmDriverExtra fields for allocating/storing/manipulating the structure. These fields define the sizes of the public and private parts of the structure, respectively.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprint
KBSubcategory: GdiPrn

Dialog Box Frame Styles

PSS ID Number: Q74334

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Dialog boxes can have either the WS_DLGFRAME or the WS_BORDER style. If a dialog box is created with both of these styles, it will have a caption bar instead of the expected frame and border. This is because WS_BORDER | WS_DLGFRAME is equal to WS_CAPTION.

To create a dialog box with a modal dialog frame and a caption, use DS_MODALFRAME combined with WS_CAPTION.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

DIALOGEX Resource Template Differences from DIALOG Template

PSS ID Number: Q141201

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95

SUMMARY

When you create or modify a dialog template in memory, it is necessary to know the form of the DLGTEMPLATE and DLGITEMTEMPLATE structures. While these structures are well documented for DIALOG resources, they take a different form for DIALOGEX resources, and are not well documented in the SDK.

MORE INFORMATION

The extended DLGTEMPLATE structure, call it DLGTEMPLATEEX, has this form:

```
typedef struct tagDLGTEMPLATEEX{
    WORD wDlgVer;           // use 1
    WORD wSignature;        // use 0xFFFF
    DWORD dwHelpID;         // Dialog's context help ID
    DWORD dwExStyle;        // Extended style
    DWORD dwStyle;          // Style
    WORD cDlgItems;         // Number of controls in dialog
    short x;                // Initial position, horizontal
    short y;                // Initial position, vertical
    short cx;               // Width
    short cy;               // Height
} DLGTEMPLATEEX;
```

This is followed by menu name, class name, title, and font info if style includes DS_SETFONT, which have the same form as documented in the SDK for DLGTEMPLATE.

The extended DLGITEMTEMPLATE structure, call it DLGITEMTEMPLATEEX, has this form:

```
typedef struct tagDLGITEMTEMPLATEEX{
    DWORD dwHelpID;         // Context help ID for control
    DWORD dwExStyle;        // Control extended styles
    DWORD dwStyle;          // Style
    short x;                // Initial position, horizontal
    short y;                // Initial position, vertical
    short cx;               // Width
    short cy;               // Height
    DWORD dwID;             // Window ID
```

```
} DLGITEMTEMPLATEEX;
```

This is followed by class name, title, and creation data for the control, which have the same form as documented in the SDK for DLGITEMTEMPLATE.

Code Sample

The following code creates a DIALOGEX resource in memory with a button and a custom control to which it passes creation data:

```
/* allocate some memory */
pdlgtemplate = p = (PWORD) LocalAlloc (LPTR, 1000);

/* start to fill in the dlgtemplate information, addressing by WORDs */
lStyle = DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU| DS_SETFONT;
*p++ = 1;          // DlgVer
*p++ = 0xFFFF;     // Signature
*p++ = 0;          // LOWORD HelpID
*p++ = 0;          // HIWORD HelpID
*p++ = 0;          // LOWORD (lExtendedStyle)
*p++ = 0;          // HIWORD (lExtendedStyle)
*p++ = LOWORD (lStyle);
*p++ = HIWORD (lStyle);
*p++ = 2;          // NumberOfItems
*p++ = 210;        // x
*p++ = 10;         // y
*p++ = 100;        // cx
*p++ = 100;        // cy
*p++ = 0;          // Menu
*p++ = 0;          // Class

/* copy the title of the dialog */
nchar = nCopyAnsiToWideChar (p, TEXT("Dialog"));
p += nchar;

/* Font information because of DS_SETFONT */
*p++ = 18;     // point size
nchar = nCopyAnsiToWideChar (p, TEXT("Times New Roman"));  // Face name
p += nchar;

/* make sure the first item starts on a DWORD boundary */
p = lpwAlign (p);

/* now start with the first item */
lStyle = BS_PUSHBUTTON | WS_VISIBLE | WS_CHILD | WS_TABSTOP;
*p++ = 0;          // LOWORD (lHelpID)
*p++ = 0;          // HIWORD (lHelpID)
*p++ = 0;          // LOWORD (lExtendedStyle)
*p++ = 0;          // HIWORD (lExtendedStyle)
*p++ = LOWORD (lStyle);
*p++ = HIWORD (lStyle);
*p++ = 10;         // x
*p++ = 60;         // y
```

```
*p++ = 80;          // cx
*p++ = 20;          // cy
*p++ = IDOK;        // LOWORD (Control ID)
*p++ = 0;       // HOWORD (Control ID)

/* fill in class i.d., this time by name */
char = nCopyAnsiToWideChar (p, TEXT("BUTTON"));
p += nchar;

/* copy the text of the first item */
nchar = nCopyAnsiToWideChar (p, TEXT("OK"));
p += nchar;

*p++ = 0;  // advance pointer over nExtraStuff WORD

/* make sure the second item starts on a DWORD boundary */
p = lpwAlign (p);

lStyle = WS_VISIBLE | WS_CHILD;
*p++ = 0;              // LOWORD (lHelpID)
*p++ = 0;              // HIWORD (lHelpID)
*p++ = 0;              // LOWORD (lExtendedStyle)
*p++ = 0;              // HIWORD (lExtendedStyle)
*p++ = LOWORD (lStyle);
*p++ = HIWORD (lStyle);
*p++ = 20;             // x
*p++ = 5;              // y
*p++ = 65;             // cx
*p++ = 45;             // cy
*p++ = 57;             // LOWORD (Control ID)
*p++ = 0;                // HOWORD (Control ID)

// The class name of the custom control
nchar = nCopyAnsiToWideChar (p, TEXT("ACustomControl"));
p += nchar;

/* copy the text of the second item, null terminate the string. */
nchar = nCopyAnsiToWideChar (p, TEXT(""));
p += nchar;

*p++ = 8;  // number of bytes of extra data

*p++ = 0xA1;   //extra data
*p++ = 0xA2;
*p++ = 0xA3;
*p++ = 0xA4;

DialogBoxIndirect (ghInst, (LPDLGTEMPLATE) pdlgtemplate, hwnd,
(DLGPROC) About);
LocalFree (LocalHandle (pdlgtemplate));


////////////////////////////////////////////////////////////////////////
//
//
```

```
Helper routines taken from the WIN32SDK DYNDLG sample
////////////////////////////////////////////////////////////////////////
//
//
LPWORD lpwAlign ( LPWORD lpIn)
{
  ULONG ul;

  ul = (ULONG) lpIn;
  ul +=3;
  ul >>=2;
  ul <<=2;
  return (LPWORD) ul;
}

int nCopyAnsiToWideChar (LPWORD lpWCStr, LPSTR lpAnsiIn)
{
  int nChar = 0;

  do {
    *lpWCStr++ = (WORD) *lpAnsiIn;
    nChar++;
  } while (*lpAnsiIn++);

  return nChar;
}
```

REFERENCE
=========

For further information on using Dialog templates, please see the SDK documentation for the DLGTEMPLATE and DLGITEMTEMPLATE structures.

Additional reference words: 4.00 kbinf
KBCategory: kbprg kbcode
KBSubcategory:

DIB_PAL_INDICES and CBM_CREATEDIB Not Supported in Win32s

PSS ID Number: Q108497

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2
- Microsoft Win32 SDK (Windows 95 only)

The Windows NT implementation of the Win32 application programming interface (API) includes two new flags, DIB_PAL_INDICES and CBM_CREATEDIB, that can be used with various device-independent bitmap (DIB) APIs. These flags are not supported by Windows 3.1, Win32s, or Windows 95.

DIB_PAL_INDICES

In Windows NT, the DIB_PAL_INDICES flag can be used with the following APIs:

```
CreateDIBitmap()
CreateDIBPatternBrush()
CreateDIBPatternBrushPt()
SetDIBits()
GetDIBits()
SetDIBitsToDevice()
StretchDIBits()
```

When the dwUsage parameter is DIB_PAL_INDICES, the associated DIB does not have a color table. In this case, the bitmap bits are indices into the device palette.

Applications written to run on Windows 3.1 or Win32s should use DIB_PAL_COLORS or DIB_RGB_COLORS instead of DIB_PAL_INDICES.

Windows 95 does not support DIB_PAL_INDICES.

CBM_CREATEDIB

In Windows 3.1, CreateDIBitmap() creates a device-dependent bitmap (DDB) from a DIB definition and optionally initializes the DDB. In Windows NT, the CBM_CREATEDIB flag can be used with CreateDIBitmap() to create a new DIB instead of a DDB.

This functionality is not present in Windows 3.1 or Win32s.

Windows 95 does not support CBM_CREATEDIB. Equivalent functionality is provided in Windows 95 by CreateDIBSection().

Additional reference words: 1.00 1.10 1.20 4.00
KBCategory: kbprg
KBSubcategory: GdiBmp W32s

Differences Between hInstance on Win 3.1 and Windows NT

PSS ID Number: Q103644

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In Microsoft Windows version 3.1, an instance handle can be used to uniquely identify the instance of an application because instance handles are unique within the scope of an address space. Because each instance of an application runs in its own address space on Windows NT, instance handles cannot be used to uniquely identify an instance of an application running on the system. This article explains why, and some alternative calls that might assist in uniquely identifying an application instance on Windows NT.

MORE INFORMATION

Although the concepts for an instance handle are similar on Windows NT and Windows 3.1, the results you see regarding them might be very different from what you expect.

With Windows 3.1, when you start several instances of the same application, they all share the same address space. You have multiple instances of the same code segment; however, each of these instances has a unique data segment associated with it. Using an instance handle (hInstance) is a way to uniquely identify these different instances and data segments in the address space.

Instance handles are unique to the address space. On Windows NT, when looking at the value of the instance handle, or the value returned from GetWindowLong(hWnd, GWL_HINSTANCE), a developer coming from a Windows 3.1 background might be surprised to see that most of the windows on the desktop return the same value. This is because the return value is the hInstance for the instance of the application, which is running it its own address space. (An interesting side note: The hInstance value is the base address where the application's module was able to load; either the default address or the fixed up address.)

On Windows NT, running several instances of the same application causes the instances to start and run in their own separate address space. To emphasize the difference: multiple instances of the same application on Windows 3.1 run in the same address space; in Windows NT, each instance has its own, separate address space. Using an instance handle to uniquely identify an application instance, as is possible on Windows 3.1, does not apply in Windows NT. (Another interesting side note: Remember that even if

there are multiple instances of an application, if they are able to load at their default virtual address spaces, the virtual address pages of the different applications' executable code will map to the same physical memory pages.)

In Windows NT, instance handles are not unique in the global scope of the system; however, window handles, thread IDs, and process IDs are. Here are some calls that may assist in alternative methods to uniquely identify instance of applications on Windows NT:

- GetWindowThreadProcessID() retrieves the identifier of the thread that created the given window and, optionally, the identifier of the process that created the window.

- OpenProcess() returns a handle to a process specified by a process ID.

- GetCurrentProcessID() returns the calling process's ID.

- EnumThreadWindows() returns all of the windows associated with a thread.

- The FindWindow() function retrieves the handle of the top-level window specified by class name and window name.

- To enumerate all of the processes on the system, you can query the Registry using RegQueryValueEx() with key HKEY_PERFORMANCE_DATA, and the Registry database index associated with the database string "Process".

For further details on using these calls, please see the Win32 SDK help file.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMisc

Differences Between the Win32 3.5 SDK and Visual C++ 2.0

PSS ID Number: Q125474

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 3.5
- Microsoft Visual C++ 32-bit Edition, version 2.0

SUMMARY

Microsoft Visual C++ version 2.0 contains the compiler tools, headers, and libraries necessary to develop Win32-based applications. In addition, it has an integrated development environment (Microsoft Foundation Classes (MFC) version 3.0) and Wizards to make programming easier. However, there is still a separate Win32 SDK available through MSDN Level II.

This article lists the things that are included in the Win32 SDK that are not included in Visual C++ version 2.0.

MORE INFORMATION

The following is a list of items included in the retail Win32 SDK that are not included in Visual C++ version 2.0 (Intel only):

Tools

WinDbg/WinDbgRM
Process Walker
Working Set Tuner
Software Compatibility Test
Microsoft Test
API Profilers
POSIX headers and libraries
Help Indexing
masm386

Toolkits

RPC Toolkit (and samples)
Setup Toolkit

Documentation

SNMP Programmer's Reference (PROGREF.RTF)
Generic Thunks (GENTHUNK.TXT)
Multicast Extensions to Windows Sockets for Win32
Windows Sockets for Appletalk
File Formats (CUSTCNTL.TXT, ENHMETA.HLP, PE.TXT, RESFMT.TXT)

Writing Great 32-bit Applications for Windows
POSIX Conformance Document
Microsoft Windows NT Version 3.5 Hardware Compatibility List

Additional Samples

Win32:

BNDBUF	NTSD		
CDTEST	RASBERRY		
CPL	REBOOT		
DYNDLG	RNR		
GLOBCHAT	SD_FLPPY		
INTEROP	SERVENUM		
IOCOMP	SIMPLEX		
IPXCHAT	SNMP		
LARGEINT	SOCKETS		
MANDEL	SPINCUBE		
MSGTABLE	WDBGEXTS		
NETDDE			

Multimedia:

AVIEDI32	MIXAPP	
AVIVIEW	MOVPLAY	
CAPTEST	MPLAY	
DSEQFILE	PALMAP	
ICMAPP	REVERSE	
ICMWALK	TEXTOUT	
LANGPLAY	WAVEFILE	
MCIPLAY	WRITEAVI	
MCIPUZZL		

SDK Tools:

ANIEDIT	RSHELL
FONTEDIT	TLIST
IMAGE	UCONVERT
IMAGEDIT	WALKER
NETWATCH	WINAT
REMOTE	

Additional reference words: 2.00 3.50
KBCategory: kbtool
KBSubcategory: TlsMisc

Differences Between Windows 95 and Windows NT SNMP

PSS ID Number: Q139462

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1, 3.5, 3.51 and Windows 95

SUMMARY

Windows 95 and Windows NT differ in their support for SNMP. This article describes these differences.

MORE INFORMATION

The Win32 SDK supports two sets of APIs for developing SNMP applications. One API set is for developing extension agents, and the other is for writing SNMP manager applications (Management API). Windows NT supports both sets. Under Windows 95, you can write SNMP extension agents but not SNMP manager applications.

Windows NT supports the Internet-II MIB (MIB_II.MIB) as well as the LanManager MIB (LMMIB2.MIB). Windows 95 supports only the Internet-II MIB.

Windows NT allows configuration of the SNMP agent through the User Interface in the Control Panel by using the Networks icon. Windows 95 does not allow this configuration. To configure the Windows 95 agent, users must modify the registry directly.

REFERENCES

Win32 SDK Compact Disc \Docs\Misc\Progref.rtf
Windows 95 Resource Kit

Additional reference words: 4.00 3.10 3.50 3.51
KBCategory: kbnetwork
KBSubcategory:

Differentiating Between the Two ENTER Keys

PSS ID Number: Q77550

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

An application may find it useful to differentiate between the user pressing the ENTER key on the standard keyboard and the ENTER key on the numeric keypad. Either action creates a WM_KEYDOWN message and a WM_KEYUP message with wParam set to the virtual key code VK_RETURN. When the application passes these messages to TranslateMessage, the application receives a WM_CHAR message with wParam set to the corresponding ASCII code 13.

To differentiate between the two ENTER keys, test bit 24 of lParam sent with the three messages listed above. Bit 24 is set to 1 if the key is an extended key; otherwise, bit 24 is set to 0 (zero). The contents of lParam for these messages is documented in the "Microsoft Windows Software Development Kit Reference Volume 1" for version 3.0 of the SDK and in the SDK Reference Volume 3, "Messages, Structures, and Macros."

Because the keys in the numeric keypad (along with the function keys) are extended keys, pressing ENTER on the numeric keypad results in bit 24 of lParam being set, while pressing the ENTER key on the standard keyboard results in bit 24 clear.

The following code sample demonstrates differentiating between these two ENTER keys:

```
   case WM_KEYDOWN:
      if (wParam == VK_RETURN)    // ENTER pressed
         if (lParam & 0x1000000L) // Test bit 24 of lParam
            {
            // ENTER on numeric keypad
            }
         else
            {
            // ENTER on the standard keyboard
            }
      break;
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95

KBCategory: kbui kbcode

KBSubcategory: UsrInp

Direct Drive Access Under Win32

PSS ID Number: Q100027

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

To open a physical hard drive for direct disk access (raw I/O) in a Win32-based application, use a device name of the form

```
\\.\PhysicalDriveN
```

where N is 0, 1, 2, and so forth, representing each of the physical drives in the system.

To open a logical drive, direct access is of the form

```
\\.\X:
```

where X: is a hard-drive partition letter, floppy disk drive, or CD-ROM drive.

MORE INFORMATION

================

You can open a physical or logical drive using the CreateFile() application programming interface (API) with these device names provided that you have the appropriate access rights to the drive (that is, you must be an administrator). You must use both the CreateFile() FILE_SHARE_READ and FILE_SHARE_WRITE flags to gain access to the drive.

Once the logical or physical drive has been opened, you can then perform direct I/O to the data on the entire drive. When performing direct disk I/O, you must seek, read, and write in multiples of sector sizes of the device and on sector boundaries. Call DeviceIoControl() using IOCTL_DISK_GET_DRIVE_GEOMETRY to get the bytes per sector, number of sectors, sectors per track, and so forth, so that you can compute the size of the buffer that you will need.

Note that a Win32-based application cannot open a file by using internal Windows NT object names; for example, attempting to open a CD-ROM drive by opening

```
\Device\CdRom0
```

does not work because this is not a valid Win32 device name. An application can use the QueryDosDevice() API to get a list of all valid Win32 device names and see the mapping between a particular Win32 device name and an

internal Windows NT object name. An application running at a sufficient privilege level can define, redefine, or delete Win32 device mappings by calling the DefineDosDevice() API.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

Disabling the Mnemonic on a Disabled Static Text Control

PSS ID Number: Q66946

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Disabling a static text control that contains a mnemonic does not prevent the control from responding to that key. For more information on static text controls with mnemonics, query in this Knowledge Base on the word "mnemonic".

To keep a static text control from processing mnemonics, it must be subclassed. When the control is disabled, WM_GETTEXT messages must be intercepted and the subclass procedure should return an empty string in response to the message.

MORE INFORMATION
================

When a static text control with a mnemonic is disabled using EnableWindow(), it turns gray but it does not stop responding to the mnemonic. This can cause problems because Windows processes the mnemonic by setting the focus to the next nonstatic, enabled control.

It is necessary to resort to subclassing to prevent the static control from processing the mnemonic. The subclass procedure should process the WM_GETTEXT message as follows:

```
    ...
    // Windows asks the control, hChild, for its text.
    case WM_GETTEXT:
        if (!IsWindowEnabled(hChild))
            {
            *(LPSTR)(lParam) = 0;    // A null terminated empty string
            return 0L;
            }
        break;
    ....
```

When the ALT key is held down and a key is pressed, Windows scans the text of each control to see if the key corresponds to a mnemonic. A WM_GETTEXT message is sent to each control. Normally, the control processes this message by returning its text. By returning an empty string in response to this message, Windows does not find the mnemonic.

Because the mnemonic must work when the control is enabled, the IsWindowEnabled() function is used to determine the state of the control. If it is enabled, default processing occurs. Otherwise, the control is disabled and no text is returned, effectively disabling the mnemonic.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Displaying on the Screen What Will Print

PSS ID Number: Q22553

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

It is possible for an application to closely simulate on the screen how an block of text will appear when printed. This article provides some references and techniques to accomplish this goal.

NOTE: WYSIWYG functionality can be attained in Windows 3.1 by using TrueType fonts. The following information should be used with non-TrueType fonts.

MORE INFORMATION
================

To create a what-you-see-is-what-you-get (WYSIWYG) screen image, the application must combine the services of the Windows graphics device interface (GDI), the printer, the display, and fonts. The following three references provide information that may be helpful:

1. "Pocket Pal -- A Graphic Arts Production Handbook" from International Paper Company; discusses issues related to type and typesetting.

2. "Bookmaking" by Marsha Lee; discusses issues related specifically to making books.

3. "Phototypesetting, A Design Manual" by James Craig.

These books contain information about type, fonts, and stringing characters together to make text. The following are three points to remember:

1. Printer manufacturers produce excellent fonts on their printers. Use printer fonts as much as possible.

2. GDI must provide some fonts for use with font-deficient display drivers.

3. Displays (even high-end ones) have much lower resolution than

printers.

The best displays attached to microcomputers are approximately 120 dots-per-inch (dpi), while the typical laser printer is 300 dpi. To produce the same size image (for example, a 9-point capital A), the printer will illuminate/paint more of its pixels than will the screen. In particular, the printer will more closely match the requested size than the display. This leads to integer round-off errors.

Most graphics output devices are raster devices. Due to integer round-off errors associated with sampling the ideal "A" for different device resolutions, the origin of characters and words and the ends of lines on the display will seldom be the same as on a printer. For example, using a 75-dpi display and 300-dpi printer, the display might choose a 6 pixel-width for the character "A", while the printer might choose a 25 or 23 pixel-width for that same character. This mismatch necessitates adjusting the text on the display to match the output on the printer.

GDI provides various approaches to find the information needed to perform the adjustments. The following applications perform this task with increasing sophistication:

1. Windows Notepad application (does a less-than-ideal job)

2. Windows Write application (does better; however, the point at which various screen lines word wrap with different fonts is jagged)

3. Word for Windows (does an excellent job of handling text)

The first two applications are included with the Windows operating environment.

When Windows starts up, GDI asks each device whether it can support any fonts. For devices that provide some intrinsic (driver-based) fonts, GDI enumerates the available fonts and creates a table that describes them. When an application requests a font from GDI (using the CreateFont and SelectObject functions), GDI selects the font in its table that most closely matches the requested font. If no device font matches well, GDI will attempt to use one of its own fonts.

Ideally, the requested font will be available for all devices. More realistically, GDI provides a similar font, within the limits of the device capabilities.

The best way to imitate the appearance of printer fonts on a display is to assume that the printer has more fonts and greater resolution than the display. The following nine steps describe one way to implement a WYSIWYG display:

1. Open a device context (DC) and enumerate the fonts available on the printer. Use information from the GetDeviceCaps function with the TEXTCAPS parameter to determine how the device can alter the appearance of the fonts it provides. Together, the EnumFonts and GetDeviceCaps functions will allow the application to determine

which fonts the device and GDI can provide. The text capabilities of the device serve as a filter in the enumeration process.

2. Provide a user interface in the application to allow the user to choose one of the fonts (this will result in "printer-centered WYSIWYG"). If appropriate, provide a method to choose between sizes and other attributes (bold, italic, and so forth). Fonts are most commonly selected by point size. One point equals 1/72 of an inch. Enumerating fonts returns LOGFONT and TEXTMETRIC structures. The quantities in these structures are in logical units that depend on the current mapping mode. Assuming that MM_TEXT (the default mapping mode) is selected, one logical unit equals one device unit (pixel, or pel). Font height and internal leading define the point size of the font as follows:

$$\text{point\_size} = \frac{72 * (\text{tmHeight} - \text{tmInternalLeading})}{\text{GetDeviceCaps(hDC, LOGPIXELSY)}}$$

3. Create a LOGFONT structure for the selected font. To ensure successful selection of that font into the printer DC, the lfHeight, lfWeight, and lfFacename fields must be specified. Weight and face name can be copied directly from the LOGFONT structure that was returned during the enumeration. The height should be computed using the following formula:

   ```
   lfHeight = -(point_size * GetDeviceCaps(hDC, LOGPIXELSY) / 72)
   ```

   In normal situations, set lfWidth to zero.

4. Once a font has been chosen, select it into the printer DC. Use GetTextMetrics and GetTextFace to verify the selection. If the steps above are followed, the process should fail only when very little memory is available in the system.

5. Create a device context for the display. Use the equation listed in step 2 above to compute the logical height for the font to be selected into the screen DC. Use CreateFont and SelectObject to select this logical font into the display DC, and use GetTextMetrics and GetTextFace to retrieve a TEXTMETRIC data structure and the face name for the selected font.

6. The font selected for the display is generally not the same as the font selected for the printer. To achieve WYSIWYG and the highest possible quality of the printed output, it is best to perform all page layout computations based on the metrics obtained from the printer DC, and force the output on the screen to match the printer output as closely as possible. This process generally causes some degradation of quality. It is assumed in the remainder of this discussion that text quality is most important.

7. Check whether either device supports the GDI escape codes that enhance the usability of fonts. One escape code, for example, returns the width table for proportionally spaced fonts. These escapes are listed in chapter 12 of the "Microsoft Windows Software

Development Kit Reference, Volume 2." Use the escape code QUERYESCSUPPORT to discover whether a device supports a particular escape code. The width tables provide data to determine the physical extent of character strings to be sent to the printer and how to match that extent on the display.

8. If the devices do not support those GDI escapes, select the desired font into a printer DC and use the GetTextExtent function to get the extent a string will occupy when printed.

9. With the methods outlined in steps 7 and 8, the dimensions of any string can be computed for the printer device. This information is used to compute page breaks and line breaks. Once the placement and extent of a string of text have been determined on the printed page, it is possible to create a scaled image on the screen. There are three methods of forcing a match between printer and screen:

   a. Take no special action. Put the text on the screen based on screen DC text metrics. With this method, only minimal matching is obtained.

   b. Use either the GetTextExtent function or the combination of the GetCharWidths, SetTextJustification, and SetTextCharacterExtra functions to create the same line breaks and justification on the screen as on the printed page. This method uses white space (usually the space character) to stretch or shrink a string of text (action of SetTextJustification) and adds a constant value to the width of every character in the font (action of SetTextCharExtra). This method achieves reasonable WYSIWYG.

   c. Use the ExtTextOut function and pass a width array to achieve the exact placement of each and every character in the string. This method provides the highest degree of WYSIWYG; however, it also requires character placement algorithms that do not degrade the speed of text output too much.

The following functions related to this article are documented in chapter 4 of the "Microsoft Windows Software Development Kit Reference, Volume 1:"

CreateFont
CreateFontIndirect
EnumFonts
Escape
GetCharWidths
GetDeviceCaps
GetTextExtent
GetTextFace
GetTextMetrics
SelectObject
SetMappingMode
SetViewportExtent
SetViewportOrigin
SetWindowExtent
SetWindowOrigin

The LOGFONT and TEXTMETRIC data structures are documented in Chapter 7 of the SDK reference, volume 2.

The following device escape functions are documented in Chapter 12 of the SDK reference, volume 2:

   ENABLEPAIRKERNING
   ENABLERELATIVEWIDTHS
   EXTTEXTOUT
   GETEXTENDEDTEXTMETRICS
   GETEXTENTTABLE
   GETPAIRKERNTABLE
   GETTRACKKERNTABLE
   QUERYESCSUPPORT
   SETKERNTRACK

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbprint
KBSubcategory: GdiPrn

Distinguishing Between Keyboard ENTER and Keypad ENTER

PSS ID Number: Q96242

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

It is possible using ReadConsoleInput() or PeekConsole() to distinguish between the ENTER key on the main keyboard and the ENTER key on the numeric keypad. The KEY_EVENT_RECORD structure in the INPUT_RECORD structure must be used to distinguish between the two keys.

MORE INFORMATION

================

The following illustrates what the KEY_EVENT_RECORD structure is filled with after a keyboard ENTER key versus a numeric keypad ENTER key is pressed.

Keyboard ENTER Key

```
   KeyEvent.wRepeatCount     = 1
   KeyEvent.wVirtualKeyCode  = 13
   KeyEvent.wVirtualScanCode = 28
   KeyEvent.dwControlKeyState= 00000000
```

Numeric Keypad ENTER Key

```
   KeyEvent.wRepeatCount     = 1
   KeyEvent.wVirtualKeyCode  = 13
   KeyEvent.wVirtualScanCode = 28
   KeyEvent.dwControlKeyState= 00000100
```

In the case of the numeric keypad ENTER key, in dwControlKeyState, the ENHANCED_KEY bit is set.

Additional reference words: 3.10 3.50 4.00 95

KBCategory: kbprg

KBSubcategory: BseCon

DLC Information on LLC_DIR_SET_MULTICAST_ADDRESS Command

PSS ID Number: Q129022

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY

The DLC programming interface for Windows NT supports Ethernet multicast addressing via the LLC_DIR_SET_MULTICAST_ADDRESS command. This command must be successfully issued before Ethernet multicasts can be received. This article shows by example how to use the command.

MORE INFORMATION

The following function demonstrates the command.

Sample Code

```
BOOL DlcSetMulticastAddress( BYTE bAdapter, BYTE *pbResult, BYTE *pbAddress
)

{
    LLC_CCB Ccb;

    Ccb.uchAdapterNumber  = bAdapter;
    Ccb.uchDlcCommand     = LLC_DIR_SET_MULTICAST_ADDRESS;

    // note:  Dlc expects Ethernet addresses to be specified in the
    //        non-canonical form.  In other words, reverse the bits
    //        before passing an Ethernet address.
    //
    //        Also, the first byte of the canonical form of an
    //        Ethernet multicast address must be 0x01.

    Ccb.u.pParameterTable = (PLLC_PARMS) pbAddress;

    if(!DlcSyncCall( &Ccb ))
        return FALSE;
    else
    {
        *pbResult = Ccb.uchDlcStatus;
        return TRUE;
    }
}


// AcsLan wrapper function used by DlcSetMulticastAddress
```

```
BOOL DlcSyncCall( PLLC_CCB pCcb )
{
   BOOL  fResult = FALSE;
   DWORD dwResult;

   pCcb->hCompletionEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

   if (!pCcb->hCompletionEvent)
      return FALSE;

   int iStatus = (int) AcsLan( pCcb, NULL );
   if ( iStatus != ACSLAN_STATUS_COMMAND_ACCEPTED )
      goto done;

   dwResult = WaitForSingleObject( pCcb->hCompletionEvent, INFINITE );

   if ( dwResult == WAIT_OBJECT_0 )
      fResult = TRUE;

done:
   CloseHandle( pCcb->hCompletionEvent );

   return fResult;
}
```

Additional reference words: 3.10 3.50
KBCategory: kbnetwork kbnetwork kbcode
KBSubcategory: NtwkMisc

DlgDirList on Novell Drive Doesn't Show Double Dots [..]

PSS ID Number: Q99339

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The Microsoft Windows application programming interface (API) function DlgDirList() can be used to add directories, drives, and/or files to a list box. On a Novell network with the default login script, the Windows API may not add the string "[..]" used to represent the parent directory to the list box. This is due to Novell implementation, and is not a bug in the Windows API. To make the entry appear, add the following line to the Novell login script (usually called SHELL.CFG located in the root directory of the boot drive)

```
SHOW DOTS = ON
```

MORE INFORMATION

A Novell NetWare file server does not include the directory entries dot (.) and double dot (..) as MS-DOS does. However, the NetWare shell (version 3.01 or later) can emulate these entries when applications attempt to list the files in a directory. Turning on Show Dots causes problems with earlier versions of some 286-based NetWare utilities, such as BINDFIX.EXE and MAKEUSER.EXE. Make sure you upgrade these utilities if you upgrade your NetWare shell. For more information, contact your Novell dealer.

NOTE: With Novell NetWare version 3.1.1, the line SHOW DOTS=ON/OFF can be added to the NET.CFG file for the same effect.

The same behavior is shown with the API DlgDirListComboBox, and the messages LB_DIR and CB_DIR.

Additional reference words: 3.10 3.50 3.51 4.00 95 listbox
KBCategory: kbui
KBSubcategory: UsrCtl

Do Not Call the Display Driver Directly

PSS ID Number: Q77402

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY
=======

In general, a Windows-based application cannot call the Windows display driver directly to perform graphics primitives. This article details the reasons this restriction is in place.

MORE INFORMATION
================

The Windows display driver communicates with the Graphics Device Interface (GDI) to perform primitive graphics operations. The parameters of the entry points (or exported functions) in the display driver are set up according to the standard interface between GDI and the display driver. The parameters passed by GDI to the display driver are only meaningful to GDI and to the display driver. A Windows-based application has no way to obtain these parameters. For example, the parameter most-commonly passed by GDI to the display driver is a pointer to a structure called PDEVICE. Memory for this structure is allocated by GDI, and its contents are specified by the display driver during the driver's initialization. The pointer to the PDEVICE structure is private to GDI; furthermore, the structure of PDEVICE varies among display drivers.

To give another example, when a primitive is to be done to a memory bitmap, instead of passing a pointer to PDEVICE, GDI passes to the display driver a pointer to a structure; the structure is usually referred to as a physical bitmap. Note that this physical bitmap structure is also called "BITMAP"; do not confuse it with the BITMAP structure defined in the Windows Software Development Kit. Again, this physical bitmap structure is not designed to be used by a Windows-based application. Although the information described in this structure is somewhat related to the bitmap that the application uses, the pointer to the physical bitmap structure is private to GDI and cannot be obtained by the application.

Additional reference words: 3.00 3.10
KBCategory: kbgraphic
KBSubcategory: GdiMisc

Do Not Forward DDEML Messages from a Hook Procedure

PSS ID Number: Q89828

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

If an application for Windows uses the Dynamic Data Exchange Management Library (DDEML) in addition to a message hook [for example, by calling SetWindowsHook() or SetWindowsHookEx()], it is possible that your hook procedure will receive messages that are intended for the DDEML libraries.

For the DDEML libraries to work properly, you must make sure that your hook function does not forward on any messages that are intended for the DDEML libraries.

MORE INFORMATION

If your hook procedure receives a code of type MSGF_DDEMGR, you should return FALSE instead of calling the CallNextHookEx() function.

The way to handle this situation is to use the following code:

```
if (MSGF_DDEMGR == code)
   return FALSE;
else
{
...
}
```

In cases where the callback function processes the message, it should return TRUE.

Note, however, how the message filter function is called from within DDEML:

```
while (TimeOutHasntExpired) {
    GetMesage (&msg, (HWND)NULL, 0, 0);
    if ( !CallMsgFilter (&msg, MSGF_DDEMGR))
         DispatchMessage (&msg);
 }
```

Given this, a callback function that just returns would cause the

CallMsgFilter() call above to return TRUE, and never dispatch the message. This inevitably causes an infinite loop in the application, because GetMessage() ends up retrieving the same message over and over, without dispatching it to the appropriate window for processing.

Therefore, a callback function that processes the message may not just return TRUE, but should also translate and dispatch messages appropriately.

The Windows 3.1 SDK's DDEMLCL sample demonstrates how to do this correctly in its MessageFilterProc() found in DDEMLCL.C:

```
   if (nCode == MSGF_DDEMGR) {

   /*
   * If a keyboard message is for MDI, let MDI client take care of it.
   * Otherwise, check to see if it is a normal accelerator key.
   * Otherwise, just handle the message as usual.
   */

       if ( !TranslateMDISysAccel (hWndMDIClient, lpmsg) &&
             !TranslateAccelerator (hWndFrame, hAccel, lpmsg))  {
             TranslateMessage (lpmsg);
             DispatchMessage (lpmsg);
         }
        return 1;
    }
```

For more information about message hooks and DDEML, please see the above mentioned functions in the Windows SDK manual or the online help facility.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

DOCERR: AddPort, ConfigurePort, DeletePort Fail Remotely

PSS ID Number: Q131223

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS

Using AddPort, ConfigurePort, or DeletePort with the Universal Naming Convention (UNC) name of a remote server as the first parameter results in a return value of FALSE. This indicates that a failure has occurred. The error returned by GetLastError is ERROR_NOT_SUPPORTED. If the first parameter is the UNC name of the local machine or NULL, the functions succeed.

CAUSE

Calls to remote servers via AddPort, ConfigurePort and DeletePort are not supported. The documentation gives the impression that the UNC name for a remote server is permissible as the first parameter.

RESOLUTION

Use the UNC name to the local server or NULL as the first parameter.

MORE INFORMATION

These functions display a dialog associated with the hWnd that is supplied in the second parameter. Because the call displays a dialog box and requires a handle to a window in the second parameter, it is only supported locally.

Additional reference words: 3.50 4.00 95
KBCategory: kbprint kbdocerr
KBSubcategory: GdiPrn

DOCERR: BUFFER.CREATE Command Not Documented

PSS ID Number: Q123462

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 3.5

SUMMARY

This article documents the BUFFER.CREATE command. This is a command that Microsoft added, so it is not documented in IBM LAN Technical Reference.

MORE INFORMATION

The AcsLan function topic in the Win32 Help file indicates that there are some differences between Windows NT DLC and the CCB2 interface documented in the IBM LAN Technical Reference. The most notable is that the buffer pool must be specified on an open adapter instance basis, not per SAP. After an adapter has been opened by using the DIR.OPEN.ADAPTER command, a buffer pool must be given to the DLC driver by using BUFFER.CREATE.

The IBM LAN Technical Reference manual has no reference to BUFFER.CREATE. However, the DLCAPI.H file contains the following structure with no other information explaining how to use it:

```
typedef struct {
    HANDLE hBufferPool;               // handle of new buffer pool
    PVOID pBuffer;                    // any buffer in memory
    ULONG cbBufferSize;               // buffer size in bytes
    ULONG cbMinimumSizeThreshold;     // minimum locked size
} LLC_BUFFER_CREATE_PARMS, *PLLC_BUFFER_CREATE_PARMS;
```

Unlike DLC as described in IBM LAN Technical Reference, Windows NT DLC does not support a buffer pool per SAP, but rather a buffer pool per process. All buffers given to the application are allocated from the same pool. (You can also allocate send buffers from the buffer pool, but this is not the case in general.)

The application gives the buffer pool to DLC by using the BUFFER.CREATE request. The parameters are:

- hBufferPool: the returned handle of the buffer pool. Supposedly, DLC can share buffer pools between processes but it doesn't, so it's useless.

- pBuffer: the pointer to application allocated buffer pool. This can be allocated by any scheme you wish and can be any size.

- cbBufferSize: the size of pBuffer.

- cbMinimumSizeThreshold: the minimum locked size.

Because the buffer is passed on kernel mode, it must be mapped into system space. It is divided into pages and further subdivided into 256-byte segments (unknown to the application). DLC tries to return as many contiguous segments as it can, but you may end up with a fragmented buffer, containing a relatively long chain of small segments. To your advantage, the largest DLC receive frame is usually around 4K.

DLC aligns the buffer to a page boundary, discarding any partial page buffer after the last page. This is not a problem unless you allocate a buffer less than a page, or one that is not page-aligned. In these cases, you are in danger of not having a buffer.

Because the buffer must be mapped to system space, DLC needs to lock the buffers into memory when receiving or sending from the buffers. To increase performance, DLC tries to keep a certain amount of buffer locked at all times. It will lock successive pages as the need arises. (In Windows NT, the smallest lockable region is a page.) You can control the initial locked area by setting the cbMinimumSizeThreshold value. If DLC does indeed lock your pages in excess of cbMinimumSizeThreshold, it will try to unlock them as they are freed up.

Additional reference words: 3.50
KBCategory: kbnetwork kbdocerr
KBSubcategory: NtwkMisc

DOCERR: CreateFile() and Mailslots

PSS ID Number: Q131493

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

SUMMARY

=======

The documentation for CreateFile() API gives incorrect possible return values while opening a client end of a mailslot. The documentation states:

> If CreateFile opens the client end of a mailslot, the function always returns a valid handle, even if the mailslot does not exist. In other words, there is no relationship between opening the client end and opening the server end of the mailslot.

Actually, CreateFile() returns INVALID_HANDLE_VALUE for a mailslot if the mailslot client is being created using the "\\." notation to communicate with a mailslot server on the local system when the server is not up and running.

MORE INFORMATION

================

The following code always returns INVALID_HANDLE_VALUE for a handle value from CreateFile() while opening the client end of the mailslot if the mailslot server is not up and running to read mailslot messages:

```
CreateFile("\\\\.\\mailslot\\testslot",
           GENERIC_WRITE,
           FILE_SHARE_READ,
           NULL,
           OPEN_EXISTING,
           FILE_ATTRIBUTE_NORMAL,
           NULL);
```

GetLastError() in this case returns Error 2: "The system cannot find the specified file."

This is an expected behavior. Windows NT implementation of local mailslots does not allow you to open the mailslot if the receiver has not created the server end with CreateMailslot() API.

Additional reference words: 3.50

KBCategory: kbprg kbnetwork kbdocerr

KBSubcategory: BseIpc

DOCERR: CS_PARENTDC Class Style Description Incorrect

PSS ID Number: Q111005

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The documentation relating to the CS_PARENTDC window class style in the Windows SDK versions 3.0 and 3.1 states that CS_PARENTDC "Gives the display context of the parent window to the window class." The documentation for the Win32 SDK contains a similar statement. These statements are incorrect.

MORE INFORMATION

A window with the CS_PARENTDC style bit will receive a regular device context (DC) from the system's cache of device contexts. CS_PARENTDC merely sets the clipping rectangle of the child to that of the parent so that the child can draw on the parent. It does not give the child the parent's DC or DC settings.

CS_PARENTDC is used with all of Windows's standard controls such as edit controls and list boxes because it can help improve performance when drawing, especially in a dialog box. For example, dialog box units are not exact and if a child is drawing in a very small space it might accidentally draw outside its own border. This style provides a little room for error and prevents the child from being clipped unexpectedly.

Additional reference words: 3.00 3.10 3.50 4.00 95 WNDCLASS RegisterClass
KBCategory: kbui kbdocerr
KBSubcategory: UsrCls

DOCERR: Custom Input/Output Procedure for Compound Files

PSS ID Number: Q140429

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 4.0

SUMMARY
=======

The first paragraph of the "Using Custom Input/Output Procedures" topic in the Win32 SDK "Overviews, Multimedia Services, Multimedia File Input/Output" section refers to a predefined I/O procedure for accessing compound files. This reference has nothing to do with OLE compound files and should be disregarded. It refers to a completely different and completely obsolete file format. There is no predefined I/O procedure for OLE compound files. Please see the documentation for the products listed at the beginning of this article for information about writing a custom I/O procedure.

MORE INFORMATION
================

The incorrect paragraph from the Win32 SDK appears below. The "Multimedia Programmer's Guide" from the Windows 3.1 SDK contains a similar paragraph under the "Using Custom I/O Procedures" topic.

Multimedia file I/O services use I/O procedures to handle the physical input and output associated with reading and writing to different types of storage systems, such as file-archival systems and database-storage systems. There are predefined I/O procedures for standard Microsoft Windows files and for memory files. In the future, there will be a predefined I/O procedure for accessing elements of compound files. Compound files consist of a number of individual files, called file elements, bound together in one physical file.

Additional reference words: 3.10 3.51 4.00 mmioInstallIOProc mmioSendMessage mmioOpen IOProc docerr
KBCategory: kbmm kbprg kbdocerr
KBSubcategory: MMIO

DOCERR: DdeCreateStringHandle() lpszString param

PSS ID Number: Q102570

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The documentation for the DdeCreateStringHandle() function in the Windows 3.1 Software Development Kit (SDK) "Programmer's Reference, Volume 2: Functions" incorrectly states that the lpszString parameter may point to a buffer of a null-terminated string of any length, when the string is actually limited to 255 characters.

DDEML string-management functions are internally implemented using global atom functions. DdeCreateStringHandle() in particular, internally calls GlobalAddAtom(), and therefore inherits the same limitation as atoms to a maximum of 255 characters in length.

MORE INFORMATION
================

DDEML applications use string handles extensively to carry out DDE tasks. To obtain a string handle for a string, an application calls DdeCreateStringHandle(). This function registers the string with the system by adding the string to the global atom table, and returns a unique value identifying the string.

The global atom table table in Windows can maintain strings that are less than or equal to 255 characters in length. Any attempt to add a string of greater length to this global atom table will fail. Hence, a call to DdeCreateStringHandle() fails for strings over 255 characters long.

This limitation is by design. DDEML applications that use the DdeCreateStringHandle() function should conform to the 255-character limit.

This limitation has been preserved on the Windows NT version of DDEML for compatibility reasons.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbdocerr
KBSubcategory: UsrDde

DOCERR: DeviceIoControl Requires OVERLAPPED Struct w/ Async.

PSS ID Number: Q126282

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The documentation for DeviceIoControl is incomplete. It should warn you that unexpected behavior may occur when both the following are true:

- A DeviceIoControl request is followed by another DeviceIoControl request on the same handle.

- There was no OVERLAPPED structure passed into DeviceIoControl after the handle to the device or file was opened with FILE_FLAG_OVERLAPPED.

The following is a list of some of the unexpected behavior that may occur:

- Requests complete before expected.

- There's undefined data in the returned buffers.

- Unknown error codes are returned.

- IRPs are held in a driver while the DeviceIoControl call has already returned.

When a handle is requested to a driver or file with FILE_FLAG_OVERLAPPED specified, the executive prepares itself for all requests on that handle to be asynchronous. Usually, when the request is sent down with an OVERLAPPED structure, an event is placed in that OVERLAPPED structure. This event is then stored in the FILE_OBJECT in kernel mode to use later to signal the user-mode application when that IRP has been completed. If an event is not specified, the value will be 0 in the FILE_OBJECT, when a second request is sent down before the first request completes (and completes), the IO manager will not have separate signals for the completion of the requests. Therefore, the request will appear to be completed, while in reality the IRP has not been completed by the underlying driver.

Please see the current documentation of GetOverlappedResult for more information on the behavior of the executive when no OVERLAPPED structure is specified when the handle to the driver or file was opened with FILE_FLAG_OVERLAPPED.

Additional reference words: 3.10 3.50 FILE_FLAG_OVERLAPPED DDK EVENT
KBCategory: kbprg kbdocerr
KBSubcategory: BseFileio

DOCERR: DEVMODE dmPaperSize Member Documentation Error

PSS ID Number: Q108924

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The dmPaperSize member of the DEVMODE structure is documented incorrectly. The documentation states that the dmPaperSize member may be set to zero if the length and width of the paper are specified by the dmPaperLength and dmPaperWidth members, respectively. However, the correct value to use for user-defined paper sizes is DMPAPER_USER.

DMPAPER_USER is correctly listed in the Microsoft Windows 3.1 SDK documentation as meaning a user-defined paper size, but is completely omitted from the Microsoft Windows 3.0 SDK documentation.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbprint kbdocerr
KBSubcategory: GdiPrn

DOCERR: EM_POSFROMCHAR & EM_CHARFROMPOS Documented Incorrectly

PSS ID Number: Q137805

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0
- Microsoft Win32s version 1.3

SUMMARY
=======

The Win32 SDK documentation for EM_POSFROMCHAR and EM_CHARFROMPOS is incorrect.

MORE INFORMATION
================

The definitions for EM_POSFROMCHAR and EM_CHARFROMPOS should be documented as follows:

EM_POSFROMCHAR

```
wParam = (WPARAM)pPoint;       // address of POINT structure
lParam = (LPARAM)wCharIndex;   // zero-based index of character
```

An application sends the EM_POSFROMCHAR message to retrieve the coordinates of the specified character in an edit control.

Parameters:

pPoint - Value of wParam. Specifies a pointer to a POINT structure that will receive the coordinates.

wCharIndex - Value of lParam. Specifies the zero-based index of the character.

Return Value - not used.

Remarks: For a single-line edit control, the y-coordinate is always zero. A returned coordinate can be negative if the character has been scrolled outside the edit control's client area. The coordinates are truncated to integer values and are in screen units relative to the upper-left corner of the client area of the control.

EM_CHARFROMPOS

```
wParam = 0;                 // not used
lParam = (LPARAM)&pPoint;   // address of a POINT structure
```

An application sends an EM_CHARFROMPOS message to retrieve the zero-based character index and zero-based line index of the character nearest the

specified point in an edit control.

Parameters:

   pPoint - Value of lParam. Specifies a pointer to a POINT structure that contains the coordinates for the character position wanted. The coordinates are in screen units relative to the upper-left corner of the client area of the control.

   Return Value - The return value specifies the character index.

Additional reference words: 4.00
KBCategory: kbprg kbui kbdocerr
KBSubcategory: UsrMsg

DOCERR: EofChar Field of DCB Structure Is Not Supported

PSS ID Number: Q115083

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS

The documentation for the device control block (DCB) structure says that the EofChar field specifies the value of the character used to signal the end of the data over the communications resource. However, this setting seems to have no effect.

RESOLUTION

DCB.EofChar is not currently supported in Windows NT.

MORE INFORMATION

When DCB.EofChar is supported, you should call ClearCommError() after every read to see whether COMSTAT.fEof is set. When COMSTAT.fEof is set, this means that DCB.EofChar has been received.

Additional reference words: 3.10 3.50
KBCategory: kbref kbdocerr
KBSubcategory: BseCommapi

DOCERR: GetQueuedCompletionStatus() Error Code

PSS ID Number: Q139460

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51

SYMPTOMS
========

Windows NT provide a powerful mechanism for performing file I/O asynchronously by using I/O Completion Ports. For more details, please see the Win32 SDK Help file and the Pop3srv sample.

An application may post aynchronous ReadFile and WriteFile messages and then carry on further processing. After some time it may call GetQueuedCompletionStatus() to find out the status of the ReadFile and WriteFile operations. It is possible that the pending file operation may still not have completed. In such a case, if there is a timeout associated with GetQueuedCompletionStatus(), the function will return FALSE and GetLastError() will return 258.

Typing "net helpmsg 258" returns this message:

   258 is not a valid Windows NT network message number.

RESOLUTION
==========

258 (0x102) is a WAIT_TIMEOUT. For more information, please refer to the Win32 SDK Winbase.h and Winnt.h files. Please note that this API is not available under Windows 95.

STATUS
======

This behavior is not documented as well as it should be.

Additional reference words: 3.50 3.51
KBCategory: kbnetwork kbdocerr kbtshoot
KBSubcategory: BseErrdebug BseFileio

DOCERR: GetWindowPlacement Function Always Returns an Error

PSS ID Number: Q89569

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When an application developed for the Microsoft Windows graphical environment uses the GetWindowPlacement() function to retrieve the show state and position information for a window, the function always returns FALSE, indicating an error.

CAUSE

The length member of the specified WINDOWPLACEMENT data structure is not initialized.

RESOLUTION

Initialize the length member and call the GetWindowPlacement() function as follows:

```
BOOL             bResult;
WINDOWPLACEMENT  lpWndPl;

lpWndPl.length = sizeof(WINDOWPLACEMENT);
bResult = GetWindowPlacement(hWnd, &lpWndPl);
```

MORE INFORMATION

The need to initialize the length member of the WINDOWPLACEMENT structure is documented on page 422 of the Microsoft Windows Software Development Kit (SDK) "Programmer's Reference, Volume 3: Messages, Structures, and Macros" manual for version 3.1. This information is not listed in the documentation for the GetWindowPlacement() function on page 479 of the "Programmer's Reference, Volume 2: Functions" manual.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbdocerr
KBSubcategory: UsrWndw

DOCERR: Important Information on RPC 2.0 Missing from the Docs

PSS ID Number: Q129143

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows for Workgroups version 3.11
- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SUMMARY
=======

This article contains important information not included in the Microsoft RPC programming documentation or in Help about Microsoft RPC version 2.0. This information is identical to the information provided in the RPCREAD.ME file in the Win32 SDK (MSTOOLS\SAMPLES\RPC\) for Windows NT version 3.5.

Microsoft RPC is a toolkit for developing distributed applications in C/C++. The toolkit includes:

- MIDL compiler for Microsoft Windows NT.

- C/C++ language header files and run-time libraries for Windows NT, Microsoft Windows versions 3.x, and MS-DOS.

- Sample programs for Windows NT, Windows versions 3.x, and MS-DOS.

MORE INFORMATION
================

This article is organized into the following sections:

- Installation
- Documentation
- Before Using RPC
- MIDL Issues
- Run-Time API Issues

Installation

The Windows NT SDK provides the components of the RPC toolkit as part of its standard installation. No additional installation is required. To produce MS-DOS and Windows 3.x RPC clients, install version 1.50 of the Microsoft Visual C/C++ development environment, and then install the MS-DOS/Win16 RPC tookit from disk. To install the MS-DOS/Windows 3.x version of the RPC toolkit, use the Setup program in the MSTOOLS\RPC_DOS\DISK1 directory.

Because the 16-bit MIDL compiler is no longer supported, you must develop applications using the 32-bit MIDL compiler. However, to write MS-DOS or Windows 3.x applications without a 32-bit C/C++ compiler capable of

targeting MS-DOS, you must compile the IDL file with a 32-bit MIDL compiler on Windows NT. Then, compile the application and stubs using your C/C++ compiler. To write MS-DOS or Windows 3.x applications using a 32-bit C/C++ compiler that is capable of targeting MS-DOS, compile both the IDL file and C/C++ files on Windows NT.

Documentation

The RPC Programmer's Guide and Reference is available in Help and includes conceptual material, MIDL language and command-line references, and run-time API references.

Because many new features have been included in this version of Microsoft RPC, see the New in this Version of RPC help topic.

RPC sample program source files are available in the directory MSTOOLS\SAMPLES\RPC. The MSTOOLS\SAMPLES\RPC\README.TXT file describes each sample.

Before Using RPC

Read the following section on MIDL and run-time API issues before attempting to use this version of RPC. These sections contain important information that is not documented in Help.

MIDL Issues

The 16-bit MIDL compiler for MS-DOS or Windows 3.x is no longer supported. Use the 32-bit MIDL compiler switch /env with either the DOS or WIN16 option.

Note the following when using this version of the MIDL compiler:

- When compiling the generated stubs, warnings about different levels of pointer indirection or different const specifications are benign.

- When using the DOS or WIN option of the -env switch, the compiler will not set the correct packing level to 2. In this case, you must explicitly specify the correct packing level by using the /Zp or /pack switch.

- For Alpha platforms, procedure serialization stubs must be compiled on the C compiler by using the -Od switch. The procedure encoding/decoding stubs on the Alpha platform should not be compiled as optimized (for example, using the -Ox switch). This affects the stubs on the Alpha platform only if the serializing procedure uses one of the following attributes on top-level parameters:

  - array or ptr attributes, such as size_is or length_is.
  - The switch_is union attribute.

- A structure whose only member is a conformant string results in an MIDL

compiler assert.

- In the osf mode of the compiler, the default allocate and free routines map to NdrRpcSmClientAllocate and NdrRpcSmClientFree. The signatures for these routines are in RPCNDR.H

Run-Time API Issues

When using the ncacn_spx and ncadg_ipx transports, the server name is exactly the same as the Windows NT server name. However, because the names are distributed using Novell protocols, they must conform to the Novell naming conventions. If a server name is not a valid Novell name, servers will not be able to create endpoints with the ncacn_spx or ncadg_ipx transports. The following is a partial list of characters prohibited in Novell server names:

" * + . / : ; < = > ? [ ] \ |

When using ncadg_ipx on Windows 3.x or MS-DOS platforms, use the Windows NT 3.1 model of server naming. That is, a tilde, followed by the servers eight-digit network number, followed by its twelve-digit Ethernet address.

The ncacn_spx transport is not supported by the version of NWLink supplied with MS Client 3.0; however, ncadg_ipx is supported.

Backslashes are now optional in the host name for ncacn_np, for consistency with the other transports.

The datagram protocols (ncadg_ipx, ncadg_ip_udp) have the following limitations:

- They do not support callbacks. Any functions using the callback attribute will fail.

- They do not support the RPC security API (RpcBindingSetAuthInfo, RpcImpersonateClient, and so on).

Only the ncacn_spx and ncacn_ip_tcp protocols support cancels. For all other protocols, the cancel routines will return RPC_S_OK, but there will be no effect. Specifically:

- RpcCancelThread will alert the specified thread, but will not interrupt a pending RPC.

- RpcTestCancel will return RPC_S_OK if the current thread has been alerted.

- RpcMgmtSetCancelTimeout has no visible effect.

RpcTestCancel, RpcMgmtSetCancelTimeout, and RpcCancelThread are only supported on Windows NT platforms; all other platforms return RPC_S_CANNOT_SUPPORT.

The function RpcBindingSetAuthInfo does not accept the value

RPC_C_AUTHN_DEFAULT.

To use the RpcBindingSetAuthInfo routine and, in particular, when using MS-DOS authentication in this version of RPC, note this information:

- The constant RPC_C_AUTHN_WINNT has been defined in the header RPCDCE.H. This constant should be passed as the AuthnSvc parameter to use the Windows NT LAN Manager Security Support Provider (NTLMSSP) service.

- When using the RPC_C_AUTHN_WINNT authentication service, the AuthIdentity parameter should be a pointer to a SEC_WINNT_AUTH_IDENTITY structure, defined in RPCDCE.H. This structure contains strings for the user's domain, username, and password. You can pass a NULL pointer to use the information for the currently logged-in user.

For MS-DOS, you cannot pass NULL because there is no way for RPC to determine the current user's logon information. However, either a structure or NULL is acceptable for Windows versions 3.x and Windows NT.

When using authenticated RPC on a system composed of Novell NetWare and Windows version 3.x (not Windows for Workgroups), you must run the optional NETBIOS.EXE program before starting Windows.

The 16-bit SDK cannot be installed on a Windows NT computer unless the current user is an administrator.

Supported Platforms for Runtime APIs

The following function is supported only by 16-bit Windows 3.x platforms:

RpcWinSetYieldInfo

The following functions are supported only by 32-bit Windows NT platforms:

DceErrorInqTest
RpcBindingInqAuthClient
RpcBindingServerFromClient
RpcCancelThread
RpcEpRegister
RpcEpUnregister
RpcIfIdVectorFree
RpcImpersonateClient
RpcNetworkInqProtseqs
RpcObjectInqType
RpcObjectSetInqFn
RpcObjectSetType
RpcProtseqVectorFree
RpcRevertToSelf
RpcServerInqBindings
RpcServerInqIf
RpcServerListen
RpcServerRegisterAuthInfo
RpcServerRegisterIf
RpcServerUnregisterIf

RpcServerUse*Protseq*
RpcMgmtInqIfIds
RpcMgmtInqStats
RpcMgmtIsServerListening
RpcMgmtInqServerPrincName
RpcMgmtSetAuthorizationFn
RpcMgmtSetCancelTimeout
RpcMgmtSetServerStackSize
RpcMgmtStatsVectorFree
RpcMgmtStopServerListening
RpcMgmtWaitServerListen
RpcMgmtEnableIdleCleanup
RpcMgmtEpElt*
RpcMgmtEpUnregister
RpcNsBindingExport
RpcNsBindingUnexport
RpcTestCancel

All other runtime functions are supported on Windows NT, Windows 3.x, and MS-DOS platforms.

REFERENCES
==========

RPCREAD.ME in \MSTOOLS\SAMPLES\RPC in Win32SDK for Windows NT 3.5 on MSDN.

Additional reference words: 6.20 3.50
KBCategory: kbnetwork kbtshoot kbref kbdocerr
KBSubcategory: NtwkRpc

DOCERR: Incorrect DialogBoxIndirect() Code in Win32 SDK Docs

PSS ID Number: Q140725

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:

    - Window 95
    - Windows NT version 3.51

Note: This docerr is fixed in Windows NT version 4.0 beta documents.

SUMMARY
=======

The Win32 SDK documentation demonstrates how to create a template in memory for a modal dialog box by using DialogBoxIndirect() in a section called "Creating a Template in Memory." The code included in this section of the documentation has problems in Windows 95 and Windows NT version 3.51, which may cause the dialog box to come up with only one control or may cause DialogBoxIndirect() to return -1, indicating failure.

The same problems occur if the call to DialogBoxIndirect() is replaced with CreateDialogIndirect().

MORE INFORMATION
================

There are four problems with the DialogBoxIndirect() code. The corrected code appears at the end of this article.

1. The code was intended to bring up a modal dialog box that contains a static control (an OK button) and a Help button. However, because of two conflicting lines of code that were supposed to set the DLGTEMPLATE struct's cdit member to the number of controls in the dialog box, the system is made to think there is only one control in the dialog.

   ```
   lpdt->cdit = 3;  // number of controls
   lpdt->cdit = 1;
   ```

   As is, the code works when pasted into an application, although the dialog box comes up with only the OK button and the other two controls are not shown. The second line should be removed because the dialog box actually contains three controls, not one. However, doing this while leaving the rest of the code as is causes no dialog box to come up and DialogBoxIndirect() returns -1 as a result of two other problems described in this article.

2. The DLGTEMPLATE structure passed to the DialogBoxIndirect or CreateDialogIndirect() functions is followed by one or more DLGITEMTEMPLATE structures. These DLGITEMTEMPLATE structures need to be DWORD aligned. Calling the following lpwAlign() function will do just that:

```
/*
   Helper routine.  Take an input pointer, return closest
   pointer that is aligned on a DWORD (4 byte) boundary.
*/
LPWORD lpwAlign ( LPWORD lpIn)
{
   ULONG ul;

   ul = (ULONG) lpIn;
   ul +=3;
   ul >>=2;
   ul <<=2;
   return (LPWORD) ul;
}
```

3. The help button and the static control have identical (x,y) coordinates as well as width and height (cx,cy):

```
lpdit->x  = 55; lpdit->y  = 10;
lpdit->cx = 40; lpdit->cy = 20;
```

This causes the static control to overlap the help button. You need to adjust these values accordingly so that they both show up in the dialog box. The following code shows how the modified code looks like so far. Note the calls to the lpwAlign function each time a DLGITEMTEMPLATE structure is added. This works fine in Windows NT version 3.51.

```
#define ID_HELP   150
#define ID_TEXT   200
LRESULT DisplayMyMessage(HINSTANCE hinst, HWND hwndOwner,
   LPSTR lpszMessage)
{

 HGLOBAL hgbl;
 LPDLGTEMPLATE lpdt;
 LPDLGITEMTEMPLATE lpdit;
 LPWORD lpw;
 LPWSTR lpwsz;
 LRESULT ret;

 hgbl = GlobalAlloc(GMEM_ZEROINIT, 1024);
 if (!hgbl)
     return -1;

 lpdt = (LPDLGTEMPLATE)GlobalLock(hgbl);

 // Define a dialog box.

 lpdt->style = WS_POPUP | WS_BORDER | WS_SYSMENU
             | DS_MODALFRAME | WS_CAPTION;
 lpdt->cdit = 3;  // number of controls
//  lpdt->cdit = 1;  // COMMENTED OUT -- unnecessary code
```

```
lpdt->x  = 10;  lpdt->y  = 10;
lpdt->cx = 100; lpdt->cy = 100;

lpw = (LPWORD) (lpdt + 1);
*lpw++ = 0;   // no menu
*lpw++ = 0;   // predefined dialog box class (by default)

lpwsz = (LPWSTR) lpw;
lstrcpyW(lpwsz, L"My Message");  // dialog title (Unicode)
lpw = (LPWORD) (lpwsz + lstrlenW(lpwsz) + 1);

//-----------------------
// Define an OK button.
//-----------------------
lpw = lpwAlign (lpw);
lpdit = (LPDLGITEMTEMPLATE) lpw;
lpdit->x  = 10; lpdit->y  = 70;
lpdit->cx = 80; lpdit->cy = 20;
lpdit->id = IDOK;  // OK button identifier
lpdit->style = WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON;

lpw = (LPWORD) (lpdit + 1);
*lpw++ = 0xFFFF;
*lpw++ = 0x0080;    // button class

lpwsz = (LPWSTR) lpw;
lstrcpyW(lpwsz, L"OK");  // button label (Unicode)
lpw = (LPWORD) (lpwsz + lstrlenW(lpwsz) + 1);
*lpw++ = 0;              // no creation data

//-----------------------
// Define a Help button.
//-----------------------
lpw = lpwAlign (lpw);

lpdit = (LPDLGITEMTEMPLATE) lpw;
lpdit->x  = 55; lpdit->y  = 10;
lpdit->cx = 40; lpdit->cy = 20;
lpdit->id = ID_HELP;    // Help button identifier
lpdit->style = WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON;

lpw = (LPWORD) (lpdit + 1);
*lpw++ = 0xFFFF;
*lpw++ = 0x0080;                // button class atom
lpwsz = (LPWSTR) lpw;
lstrcpyW(lpwsz, L"Help");       // button label (Unicode)
lpw = (LPWORD) (lpwsz + lstrlenW(lpwsz) + 1);
*lpw++ = 0;                     // no creation data

//-----------------------
// Define a static text control.
//-----------------------
lpw = lpwAlign (lpw);

lpdit = (LPDLGITEMTEMPLATE) lpw;
```

```
 lpdit->x  = 10; lpdit->y  = 10;    // Changed this from (55,10)
 lpdit->cx = 40; lpdit->cy = 20;
 lpdit->id = ID_TEXT;  // text identifier
 lpdit->style = WS_CHILD | WS_VISIBLE | SS_LEFT;

 lpw = (LPWORD) (lpdit + 1);
 *lpw++ = 0xFFFF;
 *lpw++ = 0x0082;                          // static class

 for (lpwsz = (LPWSTR)lpw;
     *lpwsz++ = (WCHAR) *lpszMessage++;
     );
 lpw = (LPWORD)lpwsz;
 *lpw++ = 0;                               // no creation data

 GlobalUnlock(hgbl);
 ret = DialogBoxIndirect(hinst,
                         (LPDLGTEMPLATE) hgbl,
                         hwndOwner, (DLGPROC) DialogProc);
 GlobalFree(hgbl);

 return ret;
}
```

4. As previously stated, this modified code works fine in Windows NT 3.51. In Windows 95, however, the dialog box and the controls come up, but with no text for the help and OK buttons or for the dialog box.

   Notice how the text in the code copies the text onto the memory block using lstrcpyW(), which is not implemented in Windows 95, so it returns ERROR_NOT_IMPLEMENTED. To generate Unicode strings in Windows 95, the application must use MultiByteToWideChar().

   Following is the modified code that works in Windows 95:

```
#define ID_HELP   150
#define ID_TEXT   200
LRESULT DisplayMyMessage(HINSTANCE hinst, HWND hwndOwner,
   LPSTR lpszMessage)
{

 HGLOBAL hgbl;
 LPDLGTEMPLATE lpdt;
 LPDLGITEMTEMPLATE lpdit;
 LPWORD lpw;
 LPWSTR lpwsz;
 LRESULT ret;
 hgbl = GlobalAlloc(GMEM_ZEROINIT, 1024);
 if (!hgbl)
     return -1;

 lpdt = (LPDLGTEMPLATE)GlobalLock(hgbl);

 // Define a dialog box.
```

```
lpdt->style = WS_POPUP | WS_BORDER | WS_SYSMENU
               | DS_MODALFRAME | WS_CAPTION;
lpdt->cdit = 3;  // number of controls
lpdt->x  = 10;  lpdt->y  = 10;
lpdt->cx = 100; lpdt->cy = 100;

lpw = (LPWORD) (lpdt + 1);
*lpw++ = 0;   // no menu
*lpw++ = 0;   // predefined dialog box class (by default)

lpwsz = (LPWSTR) lpw;
nchar = 1+ MultiByteToWideChar (CP_ACP, 0, "My Dialog", -1, lpwsz, 50);
lpw   += nchar;

//-----------------------
// Define an OK button.
//-----------------------
lpw = lpwAlign (lpw);

lpdit = (LPDLGITEMTEMPLATE) lpw;
lpdit->x  = 10; lpdit->y  = 70;
lpdit->cx = 80; lpdit->cy = 20;
lpdit->id = IDOK;  // OK button identifier
lpdit->style = WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON;

lpw = (LPWORD) (lpdit + 1);
*lpw++ = 0xFFFF;
*lpw++ = 0x0080;    // button class

lpwsz = (LPWSTR) lpw;
nchar = 1+MultiByteToWideChar (CP_ACP, 0, "OK", -1, lpwsz, 50);
lpw   += nchar;
*lpw++ = 0;              // no creation data

//-----------------------
// Define a Help button.
//-----------------------
lpw = lpwAlign (lpw);

lpdit = (LPDLGITEMTEMPLATE) lpw;
lpdit->x  = 55; lpdit->y  = 10;
lpdit->cx = 40; lpdit->cy = 20;
lpdit->id = 101; //ID_HELP;    // Help button identifier
lpdit->style = WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON;

lpw = (LPWORD) (lpdit + 1);
*lpw++ = 0xFFFF;
*lpw++ = 0x0080;                // button class atom


lpwsz = (LPWSTR) lpw;
nchar = 1+MultiByteToWideChar (CP_ACP, 0, "Help", -1, lpwsz, 50);
lpw   += nchar;
*lpw++ = 0;                     // no creation data
```

```
    //-----------------------
    // Define a static text control.
    //-----------------------
    lpw = lpwAlign (lpw);

    lpdit = (LPDLGITEMTEMPLATE) lpw;
    lpdit->x  = 10; lpdit->y  = 10;
    lpdit->cx = 40; lpdit->cy = 20;
    lpdit->id = 200; //ID_TEXT;  // text identifier
    lpdit->style = WS_CHILD | WS_VISIBLE | SS_LEFT;

    lpw = (LPWORD) (lpdit + 1);
    *lpw++ = 0xFFFF;
    *lpw++ = 0x0082;                          // static class

    for (lpwsz = (LPWSTR)lpw;
        *lpwsz++ = (WCHAR) *lpszMessage++;
       );

    lpw = (LPWORD)lpwsz;
    *lpw++ = 0;                               // no creation data

    GlobalUnlock(hgbl);
    ret = DialogBoxIndirect(hinst,
                            (LPDLGTEMPLATE) hgbl,
                            hwndOwner,
                            (DLGPROC) DialogProc);
    GlobalFree(hgbl);

    return ret;
   }
```

Additional reference words: 4.00
KBCategory: kbprg kbdocerr kbcode
KBSubcategory: UsrDlg

DOCERR: MarkMIDI Utility Not Provided in Win32 SDK

PSS ID Number: Q141087

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SUMMARY

=======

The Win32 SDK multimedia documentation refers to the MarkMIDI utility in the following topics:

   Using the MARKMIDI Utility
   Using the MCI MIDI Sequencer with High-Level Audio
   Authoring Guidelines for MIDI Files

However, this utility is not provided in the Win32 SDK.

MORE INFORMATION

================

Due to advances in the MIDI capabilities of PC sound cards and MIDI synthesizers and in particular support for the General MIDI Level 1 specification, the authoring of MIDI files according to the old Microsoft Windows 3.1 SDK Authoring Guidelines for MIDI Files is no longer recommended. Thus, the purpose for which MarkMIDI was designed (to set a flag in the header of such files) is no longer required. Also, the incorrect use of this utility can degrade the playback quality of MIDI files authored according to the General MIDI (GM) specification. Therefore it was decided to no longer include this utility in the Win32 SDK.

The information in the Win32 SDK topics regarding authoring MIDI files is obsolete information from the Windows 3.1 SDK documentation. It should be followed only if there is some special reason to author a file according to the old Windows 3.1 guidelines. In general, however, only GM files (or extensions to GM such as Roland's GS or Yamaha's XG specification) should be authored, and for these files the MarkMIDI utility must not be used.

MIDI files authored according to the old Windows 3.1 guidelines have MIDI data on channels 1 through 10, with channel 10 being percussion, and important MIDI data is duplicated on Channels 11 through 16, with 16 being percussion. For such files, the MarkMIDI utility from the Windows 3.1 SDK absolutely must be used to mark the files. The MCI sequencer (Mciseq.drv) checks for the flag set by MarkMIDI so that it knows to play back only MIDI channels 1 through 10. Without the flag, all 16 channels are played back as GM, and track 16 plays back incorrectly.

Most synthesizers and PC sound cards now support GM, which provides better sound quality than is derived from files authored according to the old Windows 3.1 guidelines. In general, there should be no reason to author MIDI files with MarkMIDI according to the old Windows 3.1 guidelines.

If a MIDI file authored according to the old Windows 3.1 guidelines is not marked, it will sound bad when played with the MCI sequencer on Windows 95. For example, if such a file included percussion on channel 16, all of that channel's data would probably be played with some instrumental sound other than percussion, and the result would be non-musical. Also, if a GM file has been marked, the MCI sequencer will only play channels 1 through 10. In that case, any MIDI data sent to channels 11 through 16 will not be heard.

REFERENCES

The General MIDI specification referred to in this article is not available from Microsoft. It is published by and is copyrighted material of the MIDI Manufacturers Association (MMA). For information on obtaining this and other MIDI specifications from the MMA, please see the following article in the Microsoft Knowledge Base:

  ARTICLE-ID: Q140203
  TITLE   : How to Obtain MIDI Specifications

Additional reference words: 4.00 3.50
KBCategory: kbmm kbprg kbdocerr
KBSubcategory: MMMidi

DOCERR: Printing an OpenGL Image

PSS ID Number: Q132866

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The documentation relating to printing an OpenGL image in the Win32 SDK versions 3.5, 3.51, and 4.0 is incorrect. The current version of Microsoft's implementation of OpenGL in Windows NT does not provide support for printing. More specifically, an application cannot call wglCreateContext or wglMakeCurrent on a printer device context.

To work around this limitation, draw the OpenGL image into a memory bitmap, and then print the bitmap.

MORE INFORMATION

Here are the correct steps to print an OpenGL image:

1. Create a printer device context.

2. Create a memory device context.

3. Create a bitmap that can be either a DIB section or a screen compatible bitmap, and select it into the memory device context. This bitmap must have four or more bits of color information per pixel.

4. Set the pixel format of the memory device context.

5. Create a rendering context, passing the wglCreateContext function the handle to the memory device context, and make that rendering context a thread's current rendering context.

6. Make OpenGL calls, which will draw into that rendering context.

7. Disconnect and delete the rendering context.

8. Use StretchDIBits, SetDIBitsToDevice, StretchBlt, or BitBlt from the memory device context to the printer device context.

9. Delete the bitmap, the memory device context, and the printer device context.

Additional reference words: 3.50 4.00 Windows 95
KBCategory: kbgraphic kbdocerr

KBSubcategory: GdiOpenGL

DOCERR: Resource Compiler Maximum String Length Incorrect

PSS ID Number: Q150448

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

The Tools User's Guide included with the Microsoft Win32 SDK specifies that string resources definined in a resource script "must be no longer than 255 characters and must occupy a single line in the source file."

This information is incorrect. The actual maximum length of a string resource for the resource compiler included with the Win32 SDK is 4097 characters.

Additional reference words: 4.00
KBCategory: kbtool kbdocerr
KBSubcategory: TlsRc

DOCERR: Return Values for EditStreamCallback Incorrect

PSS ID Number: Q136810

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

The Win32 SDK documents the return values for the EditStreamCallback incorrectly when the RichEdit control is streaming text in and out. Currently the documentation states that the return value is dependant on whether the control is reading text in or writing text out.

The correct return value for both reading and writing is zero if the read or write was successful, or an error code otherwise. The number of bytes read or written should be returned in the *pcb parameter. The return value from the EditStreamCallback is interpreted as an SCODE and zero is a successful SCODE. Whatever error code is returned by the EditStreamCallback is copied to the EDITSTREAM.dwError field so the caller can get an error status.

Additional reference words: 1.30 4.00 EM_STREAMIN EM_STREAMOUT
RichEdit
KBCategory: kbui kbdocerr
KBSubcategory: UsrCtl

DOCERR: Setting of SO_DEBUG Option Is Ignored by Winsock Apps

PSS ID Number: Q138965

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1, 3.11
- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, 4.0

The Windows Sockets specification (Winsock) version 1.1 provides for an application to set the SO_DEBUG option by calling the setsockopt() API.

Although a call to set this option succeeds (doesn't return SOCKET_ERROR), the setting is ignored by all Microsoft Windows Sockets implementations. This includes the Winsock implementations for:

- Windows for Workgroups 3.11 (MS TCP/IP-32)
- Windows NT 3.1, 3.5, 3.51
- Windows 95
- LAN Manager 2.2c

Additional reference words: 3.11 4.00 3.10 3.51 2.20c 3.50
KBCategory: kbnetwork kbdocerr
KBSubcategory: NtwkWinsock

DOCERR: SHFILEOPSTRUCT pFrom and pTo Fields Incorrect

PSS ID Number: Q133326

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

The documentation for the SHFILEOPSTRUCT structure in the Win32 SDK describes the pFrom and pTo fields incorrectly as:

> pFrom
>
> Pointer to a string that contains the names of the source files.
>
> pTo
>
> Pointer to a string that specifies the destination for the moved, copied, or renamed file.

This is only partially correct. The pFrom and pTo fields are actually pointers to a list of strings containing the desired source files. The strings should be separated by NULL characters with an extra NULL terminator after the last file name. For example, if pFrom should contain FILE1.ABC, FILE2.ABC, and FILE3.ABC, the memory pointed to by pFrom should be:

```
"FILE1.ABC\0FILE2.ABC\0FILE3.ABC\0\0"
```

If only one file is being passed in pFrom, it should still be double-NULL terminated.

Additional reference words: Win95 Shell SHFileOperation Windows 95 4.00

KBCategory: kbui kbdocerr

KBSubcategory: UsrShell

DOCERR: TabbedTextOut Tab Positions Really in Logical Units

PSS ID Number: Q113253

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

Both the printed and electronic documentation from the Windows 3.0 and 3.1 SDKs, as well as the Win32 SDK, state that the tab positions parameter of TabbedTextOut() is in device units. This is incorrect. The tab position array is in logical units. The Windows 3.0 and 3.1 SDKs refer to the tab position array as lpnTabPositions, while the Win32 SDK calls it lpnTabStopPositions.

The use of logical units can be seen by using a mapping mode other than the default MM_TEXT. The output of tabs using TabbedTextOut() reflects the mapping mode.

Additional reference words: 3.00 3.10 3.50 docerr
KBCategory: kbgraphic kbdocerr
KBSubcategory: GdiDrw

DOCERR: TranslateMessage() Always Returns True

PSS ID Number: Q137231

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The Win32.Hlp online help documentation states that the TranslateMessage() function returns TRUE only if it posts a WM_CHAR message in the thread's message queue. However, it returns TRUE on all WM_KEYDOWN and WM_KEYUP messages, regardless of translation.

This is the same behavior that Microsoft Windows 3.1 exhibits. For compatibility with Microsoft Windows 3.1, the behavior of this API call was not changed to match the documentation.

REFERENCES
==========

Win32 Software Development Kit Help file (Win32.Hlp), version 3.51; Search on: "TranslateMessage"

Additional reference words: 3.50 4.00
KBCategory: kbui kbdocerr
KBSubcategory: UsrDlgs

DOCERR: Use a Thread ID for CALLBACK_THREAD in WaveInOpen

PSS ID Number: Q134336

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

Note: This docerr has been fixed in the Windows NT 4.0 beta docs.

The Microsoft Win32 Software Development Kit (SDK) Multimedia online Help documentation incorrectly states that a thread handle should be used for the dwCallback parameter of the WaveInOpen() function when its fdwOpen parameter is CALLBACK_THREAD. Instead, the dwCallback parameter should be a thread ID. The address of the thread ID is returned as the last parameter of the CreateThread()function, which is used when the thread is created.

This documentation error occurs in the following online Help files:

Microsoft Win32 SDK Multimedia online Help included on July 1995 Microsoft Developer Network compact disc; Search on "Advanced Audio Techniques," "Waveform Audio," and "Waveform Audio Reference," Topic: "WaveInOpen."

Microsoft Win32 SDK Multimedia Reference online Help file included in Microsoft Windows 95 online documentation; Search on "WaveInOpen CALLBACK," Topic: "WaveInOpen."

Additional reference words: 3.10 3.50
KBCategory: kbmm kbsound kbdocerr
KBSubcategory: MMWave

DOCERR: Windows Sockets Enhancements Available in Windows 95

PSS ID Number: Q137484

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 4.0

The August 1995 version of the Win32 SDK incorrectly indicates that the Windows Sockets Service Registration and Resolution functions are not supported i Windows 95. These functions are supported in Windows 95. These functions include:

```
EnumProtocols
GetAddressByName
GetNameByType
GetService
GetTypeByName
SetService
```

Additional reference words: RNR
KBCategory: kbnetwork kbdocerr
KBSubcategory: NtwkWinsock Winsock

DOCERR: Winhlp32.exe Topics per Keyword is 800

PSS ID Number: Q141955

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The online documentation that ships with Help Workshop 4.0 incorrectly states that the limit for the number of topics per keyword is 64,000. The correct limit is approximately 800 topics per keyword -- although this will vary somewhat depending on the size of the keyword.

REFERENCES
==========

In Hcw.hlp, search for the "Limits" topic.

Additional reference words: 4.00
KBCategory: kbtool kbdocerr
KBSubcategory: tlshlp

Drawing a Rubber Rectangle

PSS ID Number: Q114471

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Most drawing software uses what is termed a "rubber rectangle". This term is used to describe the situation where

1. the left mouse button is held down, defining one corner of the rectangle

2. the mouse is dragged and released at the point defining the opposite corner of the rectangle

3. the rectangle is drawn while the mouse is being dragged, so that it looks like the rectangle is being stretched and contracted, like a rubber band

MORE INFORMATION
================

The key to making this work is in the following call, which should be made in the WM_LBUTTONDOWN case:

```
SetROP2( hDC, R2_NOT )
```

On each WM_MOUSEMOVE message, the rectangle is redrawn in its previous position. Because of the ROP code, the rectangle appears to be erased. The new position for the rectangle is calculated and then the rectangle is drawn.

Note that Windows will only let you draw in the invalid area of the window if you use a DC returned from BeginPaint(). If you want to use the DC returned from BeginPaint(), you must first call InvalidateRect() to specify the region to be updated.

With the DC returned from GetWindowDC(), Windows will restrict your drawing to the client and non-client areas. With the hDC returned from CreateDC(), you can write on the entire display, so you must be careful.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic

KBSubcategory: GdiDrw

Drawing Outside a Window's Client Area

PSS ID Number: Q33096

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

When an application uses the BeginPaint or GetDC function to obtain a device context (DC) for its client window and draws into this DC, Windows clips the output to the edge of the client window. While this is usually the desired effect, there are circumstances where an application draws outside the client area of its window.

MORE INFORMATION

The GetWindowDC function provides a DC that allows an application to draw anywhere within its window, including the nonclient area.

In the Windows environment, the display is a scarce resource that is shared by all applications running in the system. Most of the time, an application should restrict its output to the area of the screen it has been assigned by the user. However, an application can use the CreateDC function to obtain a DC for the entire display, as follows:

```
hDC = CreateDC("DISPLAY", NULL, NULL, NULL);
```

Device contexts are another scarce resource in the Windows environment. When an application creates a DC in response to a WM_PAINT message, it must call the DeleteDC function to free the DC before it completes processing of the message.

Painting in the nonclient area of a window is not recommended. If an application changes the nonclient area of its window, the user can become confused because the familiar Windows controls change appearance or are not available. An application should not corrupt other windows on the display.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiDc

DSKLAYT2 Does Not Preserve Tree Structure of Source Files

PSS ID Number: Q87947

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.5
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

The DSKLAYT2 utility in the Microsoft Setup Toolkit for Windows creates disk images using the files specified in the .LYT file produced by the DSKLAYT utility. The created disk images are flat; all files are placed in the root directory. DSKLAYT2 does not preserve the tree structure of the source files.

Having flat directories on the setup disks is a limitation of only the DSKLAYT2 utility. _MSTEST.EXE, the setup driver spawned by SETUP.EXE, supports a tree structure on the setup disks. Once DSKLAYT2 creates disk images, manually create subdirectories on each disk. Then move files from the root directory of the disk to the appropriate position in the tree structure. (Do not move files between disks.) Make corresponding changes to the .INF file. In the file description line of each moved file, change the filename to indicate the file's new location in the tree structure.

Additional reference words: 3.10 3.50 4.00 95 MSSetup tool kit
KBCategory: kbtool
KBSubcategory: TlsMss

Dsklayt2 Does Not Support Duplicate Filenames

PSS ID Number: Q87906

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

=======

The Dsklayt2 utility in the Microsoft Setup Toolkit for Windows does not support duplicate filenames in the source tree. If two or more files have the same name, Dsklayt2 may place more than one of these files into the root directory of the same setup disk. If this happens, Dsklayt2 generates a warning message when it creates a disk image for the disk that warns that the first file is overwritten.

If two or more files in the source tree for an application share the same name, rename all but one of the files before running the Dsklayt utility. Use the Rename Copied File option in the Dsklayt utility to have the filenames changed back to their original names when SETUP.EXE or _MSTEST.EXE copies the files from the setup disk to the destination disk.

When Dsklayt2 creates a compressed file, it adds an underscore (_) to the end of the file extension, replacing the third character if necessary. Therefore, Dsklayt2 does not support compressed files if the names of the uncompressed files differ only in the third character of the file extension.

Additional reference words: 3.10 3.50 4.00 95 MSSetup tool kit
KBCategory: kbtool
KBSubcategory: TlsMss

Dynamic Loading of Win32 DLLs

PSS ID Number: Q90745

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

When using LoadLibrary() under Win16 or OS/2, the Dynamic Link Library (DLL) is loaded only once. Therefore, the DLL has the same address in all processes. Dynamic loading of DLLs is different under Windows NT.

A DLL is loaded separately for each process because each application has its own address space, unlike Win16 and OS/2. Pages must be mapped into the address space of a process. Therefore, it is possible that the DLL is loaded at different addresses in different processes. The memory manager optimizes the loading of DLLs so that if two processes share the same pages from the same image, they will share the same physical memory.

Each DLL has a preferred base address, specified at link time. If the address space from the base address to the base address plus image size is unavailable, then the DLL is loaded elsewhere and fixups will be applied. There is no way to specify a load address at load time.

To summarize, at load time the system:

1. Examines the image and determines its preferred base address and required size.

2. Finds the address space required and maps the image, copy-on-write, from the file.

3. Applies internal fixups if the image isn't at its preferred base.

4. Fixes up all dynamic link imports by placing the correct address for each imported function in the appropriate entry of the Import Address Table. This table is contiguous with 32-bit addresses, so 1024 imports require dirtying only one page.

MORE INFORMATION

The pages containing code are shared, using a copy-on-write scheme. The term copy-on-write means that the pages are read-only; however, when a process writes the page, instead of an access violation, the memory manager makes a private copy of the page and allows the write to proceed. For example, if two processes start from the same .EXE, both initially have all pages mapped from the .EXE copy-on-write. As the two processes proceed to

modify pages, they get their own copies of the modified pages. The memory manager is free to optimize unmodified pages and actually map the same physical memory into the address space of both processes. Modified pages are swapped to/from the page file instead of the .EXE file.

There are two kinds of fixups. One is the address of an imported function. All these fixups are localized in what the Portable Executable (PE) specification calls the Import Address Table (IAT). This is an array of 32-bit function pointers, one for each imported API. The IAT is located on its own page(s), because it is always modified. Calling an imported function is actually an indirect call through the appropriate entry in this array. In case that the image is loaded at the preferred address, the only fixups needed are for imports.

Note that there is an optimization whereby each import library exports a 32-bit number for each API along with any name and ordinal. This serves as a "hint" to speed the fixups performed at load time. If the hints in the program and the DLL do not match, the loader uses a binary search by name.

The other kind of fixup is needed for references to the image's own code or data when the image can't be loaded at its preferred address. When a page must be taken out of memory, the system checks to see whether the page has been modified. If it has not, then the page is still mapped copy-on-write against the EXE and can be discarded from memory. Otherwise, it must be written to the page file before it can be removed from memory, so that the page file is used as the backing store (where the page is recovered from) rather than the executable image file.

NOTES
=====

The DLL's entry point does not get called for a second LoadLibrary() call in a process (that is, no second DLL_PROCESS_ATTACH entry). There is one call to DllEntry/DLL_THREAD_ATTACH per thread no matter the number of times a thread calls LoadLibrary(). The same goes for FreeLibrary(), but the DLL_PROCESS_DETACH happens only on the last call (that is, reference count back to zero for the process).

Global instance data for the DLL is on a per process basis (only one set per unique process). If it is necessary to ensure that global instance data is unique for each LoadLibrary() performed in a single process, consider thread local storage (TLS) as an alternative. This requires multiple threads of execution, but TLS allows unique data for each ThreadID. There is very little overhead on the DLL's part; just create a global TLS index during process initialization. During thread initialization, allocate memory (via HeapAlloc(), GlobalAlloc(), LocalAlloc(), malloc(), and so on) and store a pointer to the memory using the global TLS index value in the function TlsSetValue. Win32 internally stores each thread's pointer by TLS index and ThreadID to achieve the thread specific storage.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseDll

Efficiency of Using SendMessage Versus SendDlgItemMessage

PSS ID Number: Q66944

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

The SendDlgItemMessage function is equivalent to obtaining the handle of a dialog control using the GetDlgItem function and then calling the SendMessage function with that handle. The SendDlgItemMessage function therefore takes slightly longer to execute than the SendMessage function for the same message, because an extra call to the GetDlgItem function is required each time the SendDlgItemMessage function is called.

The GetDlgItem function searches through all controls in a given dialog box to find one that matches the given ID value. If there are many controls in a dialog box, the GetDlgItem function can be quite slow.

If an application needs to send more than one message to a dialog control at one time, it is more efficient to call the GetDlgItem function once, using the returned handle in subsequent SendMessage calls. This saves Windows from searching through all the controls each time a message is sent. The SendMessage function should also be used when your application retains handles to controls that receive messages.

However, if your application needs to send one message to many controls, such as sending WM_SETFONT messages to all the controls in a dialog, then the SendDlgItemMessage function will save code in the application because a call to the GetDlgItem function is not made for each control.

Note that if the message sent to a control may result in a lengthy operation (such as sending the LB_DIR message to a list box), then the overhead in the GetDlgItem call is negligible. Either the SendDlgItemMessage or SendMessage can be used, whichever is more convenient.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 dlgitem

KBCategory: kbui

KBSubcategory: UsrDlgs

EM_SETHANDLE and EM_GETHANDLE Messages Not Supported

PSS ID Number: Q130759

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The EM_GETHANDLE and EM_SETHANDLE messages are not supported for edit contols that are created as controls of a 32-bit application under Windows 95. This is due to the way USER is designed under Windows 95. 16-bit applications work the same way they did under Windows version 3.1. That is, they can use the EM_GET/SETHANDLE messages. Also Win32-based applications running under Windows NT will be able to use these messages.

MORE INFORMATION
================

The EM_GETHANDLE and EM_SETHANDLE messages are used to retrieve and set the handle of the memory currently allocated for a multiline edit control's text. USER under Windows 95 is a mixture of 16- and 32-bit code, so edit controls created inside a 32-bit application cannot use these messages to retrieve or set the handles. Trying to do so causes the application to cause a general protection (GP) fault and thereby be terminated by the System.

One workaround that involves a little code modification is to use the GetWindowTextLengt(), GetWindowText(), and SetWindowText() APIs to retrieve and set the text in a edit control.

NOTE: USER is almost completely 16-bit, so 32-bit applications thunk down to the 16-bit USER. Also note that the EM_GETHANDLE and EM_SETHANDLE messages cannot be used with Win32s-based applications either.

Additional reference words: 4.00 user controls GPF
KBCategory: kbui
KBSubcategory: UsrCtl

Enumerating Network Connections

PSS ID Number: Q119216

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SUMMARY
=======

From the MS-DOS prompt, you can enumerate the network connections (drives) by using the following command:

```
net use
```

Programmatically, you would call WNetOpenEnum() to start the enumeration of connected resources and WNetEnumResources() to continue the enumeration.

MORE INFORMATION
================

The following sample code enumerates the network connections:

Sample Code

```
#include <windows.h>
#include <stdio.h>

void main()
{
   DWORD dwResult;
   HANDLE hEnum;
   DWORD cbBuffer = 16384;
   DWORD cEntries = 0xFFFFFFFF;
   LPNETRESOURCE lpnrDrv;
   DWORD i;

   dwResult = WNetOpenEnum( RESOURCE_CONNECTED,
                            RESOURCETYPE_ANY,
                            0,
                            NULL,
                            &hEnum );

   if (dwResult != NO_ERROR)
   {
      printf( "\nCannot enumerate network drives.\n" );
      return;
   }

   printf( "\nNetwork drives:\n\n" );
```

```
      do
      {
         lpnrDrv = (LPNETRESOURCE) GlobalAlloc( GPTR, cbBuffer );

         dwResult = WNetEnumResource( hEnum, &cEntries, lpnrDrv, &cbBuffer
   );

         if (dwResult == NO_ERROR)
         {
            for( i = 0; i < cEntries; i++ )
            {
               if( lpnrDrv[i].lpLocalName != NULL )
               {
                  printf( "%s\t%s\n", lpnrDrv[i].lpLocalName,
                                      lpnrDrv[i].lpRemoteName );
               }
            }
         }
         else if( dwResult != ERROR_NO_MORE_ITEMS )
         {
            printf( "Cannot complete network drive enumeration" );
            GlobalFree( (HGLOBAL) lpnrDrv );
            break;
         }
         GlobalFree( (HGLOBAL) lpnrDrv );
      }
      while( dwResult != ERROR_NO_MORE_ITEMS );

      WNetCloseEnum(hEnum);
   }
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork
KBSubcategory: NtwkWinnet

ERROR_BUS_RESET May Be Benign

PSS ID Number: Q111837

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

Tape API (application programming interface) functions may return an error code of ERROR_BUS_RESET when operating on SCSI tape devices. In many cases, you can ignore this error value and retry the operation. However, this error is fatal if received during a series of write operations because a tape drive cannot recover from a bus reset and continue writing.

MORE INFORMATION

When Windows NT boots up it resets the SCSI bus. This bus reset is reported by the tape drive in response to the first operation after the reset.

The code fragment shown below in the Sample Code section could be used to check for ERROR_BUS_RESET and clear it. The same technique could be used for other informational errors, such as ERROR_MEDIA_CHANGED, that may not be relevant at application startup.

Sample Code

```
/*
** This is a code fragment only and will not compile and run as is.
*/

   ...
   do {
      dwError = GetTapeStatus(hTape);
   } while (dwError == ERROR_BUS_RESET);
   ...
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

Establishing Advise Loop on Same topic!item!format! Name

PSS ID Number: Q95983

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

Sometimes more than one DDEML client application might establish an advise loop with a server on the same topic!item!format name set. This article discusses the complexities involved in such a transaction.

MORE INFORMATION

A client application sends an XTYP_ADVSTART to DDEML when it needs periodic updates on a particular data item from a server, typically when that particular data item's value changes.

The server application calls DdePostAdvise whenever the value of the requested data item changes. This results in an XTYP_ADVREQ transaction being sent to the server's DDEML callback function, where the server returns a handle to the changed data.

The client then receives the updated data item during the XTYP_ADVDATA transaction in the case of a hot advise loop. In a warm advise loop, the XTYP_ADVDATA transaction that the client receives in its callback does not contain data; it has to specifically request data with an XTYP_REQUEST transaction.

When more than one client application requests an advise loop on the same topic!item!format name set, DDEML maintains a list of these client applications so that it knows that on one call to DdePostAdvise(), it should send the changed hData to all the requesting applications in its list.

The server's callback receives an XTYP_ADVREQ transaction as a result of DdePostAdvise() with the LOWORD (dwData1) containing a count of the number of ADVREQ transactions remaining to be processed on the same topic!item!format name set.

This count allows the server application to create its hData as HDATA_APPOWNED, whereby it could create a data handle just once and pass the same handle on to its other pending requests on the same topic!item!format name set. Finally, when the count is down to zero, DdeFreeDataHandle() can then be called on this hData.

Note that a server needs to call DdePostAdvise() only once regardless of how many pending advise requests it has on the same topic!item!format name set. This one call to DdePostAdvise() causes DDEML to send the appropriate number of XTYP_ADVREQ transactions to the server's callback.

All these can be easily illustrated using the Windows version 3.1 Software Development Kit (SDK) DDEML CLIENT and SERVER samples in this manner:

1. Start the SERVER application.

2. a. Start the CLIENT application.
   b. Establish a connection with the SERVER.
   c. Start an advise loop on the item "Rand".

3. a. Start another instance of the client application.
   b. Establish a connection with the SERVER.
   c. Start an advise loop on the item Rand.

4. Bring up DDESPY.

5. Go back to the SERVER and choose ChangeData from the Options menu and watch both CLIENT applications update their data.

Results (from DDESPY main window):

1. Two XTYP_ADVREQs (because you have two pending ADVREQs on the same test!Rand pair.

2. Changed Rand data is then sent to the first CLIENT in the advise list.

3. The first CLIENT in the advise list receives the data via XTYP_ADVDATA.

4. Changed "Rand" data is sent to the second CLIENT in the advise list.

5. The second CLIENT in the advise list receives its XTYP_ADVDATA.

One caveat to this scenario is when an advise loop is invoked with the XTYPF_ACKREQ flag set (that is, the client establishes an XTYP_ADVSTART transaction or'ed with the XTYPF_ACKREQ flag). In this case, the server does not send the next data item until an ACK is received from the client for the first data item. During a particular call to DdePostAdvise(), the server might not necessarily receive XTYP_ADVREQ in its callback for all active links, and the LOWORD(dwData1) might not necessarily reach 0 (zero). When the DDE_FACK from the client finally arrives, DDEML then sends the server an XTYP_ADVREQ with LOWORD(dwData1) set to CADV_LATEACK, identifying the late-arriving ACK appropriately.

Advise links of this kind (with XTYPF_ACKREQ flag set) are best suited to situations where the server sends information faster than a client can process it--setting the XTYPF_ACKREQ bit ensures that the server never outruns the client. However, setting this flag also sets a drawback in circumstances where data transitions may be lost. Thus, in

Windows NT or in similar situations where server outrun is highly unlikely, it is recommended that the XTYPF_ACKREQ bit not be used to prevent such data transition loss.

Note that in this delayed ACK update scenario, the count received in the LOWORD (dwData1) may not be relied upon for creating APPOWNED data handles as discussed in the earlier paragraphs; where an hData is created once, and when the count is down to zero, DdeFreeDataHandle() is called on this hData.

This does not, however, imply that the efficiency provided by APPOWNED data handles may not be used at all. In this case, a server could create an APPOWNED data handle once--usually on the first XTYP_ADVREQ it receives--and associate that handle with a topic!item!format name set. It could then return this data handle for all subsequent requests it receives on this topic!item!format set. Each time data changes thereafter, the server should destroy the old data handle and not re-render the data [that is, call DdeCreateDataHandle()] until another request comes through.

This might be better explained as follows:

```
      case XTYP_ADVREQ:
           if (ThisIsForTheTopicItemFormatSpecified)
           {
               if (bFirstTimeRequested)
                  {
                      bFirstTimeRequested = FALSE;
                      hData = DdeCreateDataHandle();
                   }
                return hData;
            }
            break;

     // and then whenever data changes for this topic!item!format
         if (hData)
           {
             DdeFreeDataHandle (hData);
             bFirstTimeRequested = TRUE;
            }
         DdePostAdvise();         // specify topic!item!format here.
                                  // This causes DDEML to send an
XTYP_ADVREQ
                                  // which is handled above.
```

For Microsoft Windows version 3.1 DDEML, the only way for a server application to distinguish which client's advise request is currently being responded to is through the XTYP_ADVREQ's hConv parameter. The hConvPartner field of the CONVINFO structure may be used to distinguish between clients.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Examining the dwOemId Value

PSS ID Number: Q101190

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

The Win32 application programming interface (API) GetSystemInfo() fills in the members of a SYSTEM_INFO structure. The dwOemId member represents a computer identifier that is specific to a particular OEM (original equipment manufacturer). Windows NT versions 3.1 - 3.51 and Windows 95 always place a zero in the dwOemId member. In later releases, this behavior will change to include different OEM IDs.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbprg
KBSubcategory: BseMisc

Explanation of the NEWCPLINFO Structure

PSS ID Number: Q103315

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The following is an explanation of the NEWCPLINFO structure:

dwSize: Specifies the length of the structure, in bytes. Set this to sizeof(NEWCPLINFO).

dwFlags: Specifies Control Panel flags. This is field is currently unused. Set this field to NULL.

dwHelpContext: Specifies the context number for the topic in the help project (.HPJ) file that displays when the user selects help for the application. If dwData is non-NULL, Windows Help will be invoked with the HELP_CONTEXT fuCommand with dwHelpContext as dwData. If dwHelpContext is NULL, Windows Help is invoked with the HELP_INDEX fuCommand.

lData: Specifies data defined by the application. This is passed back to the application in the CPL_DBLCLK, CPL_SELECT, and CPL_STOP messages via lParam2.

hIcon: Identifies an icon resource for the application icon. This icon is displayed in the Control Panel window.

szName: Specifies a null-terminated string that contains the application name. The name is the short string displayed below the application icon in the Control Panel window. The name is also displayed in the Settings menu of Control Panel.

szInfo: Specifies a null-terminated string containing the application description. The description is displayed at the bottom of the Control Panel window when the application icon is selected.

szHelpFile: Specifies a null-terminated string that contains the path of the help file, if any, for the application. If this field is unused, set it to NULL. The Control Panel will invoke a default Windows Help file when this field is NULL.

Additional reference words: 3.10 3.50 3.51 4.00 95 cpl control panel extension
KBCategory: kbui
KBSubcategory: UsrExt

Exporting Callback Functions

PSS ID Number: Q83706

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

It is not necessary for Win32-based applications to export callback functions. Windows versions 3.1 and earlier (Win16) need callback functions primarily for fixing references to global data and ensuring that EMS memory is not paged out. Neither of these situations applies to the Windows NT operating system.

MORE INFORMATION
================

Exports are necessary for any function that must be located at either

- Run time via GetProcAddress()(dynamic linking)

  -or-

- Load time via an import library (static linking)

Both of these linking methods require that the name or ordinal number of the export be known and that their names (or ordinal numbers) be present in the executable's exported entry table. This enables Windows to determine the addresses at run time.

Static linking is done by the loader, which performs this lookup for all of the imported entry points that an executable needs (normally by ordinal number). In dynamic linking, the system scans by ordinal number or by name through the DLL (Dynamic Link Library) exports table.

In Win16, exported entries are automatically fixed by the linker to adjust to the appropriate data segment. Exporting entries on Win32 just adds them to the module's exported names and ordinal numbers table; the linker does not need to "fix" them. For code compatibility with Win16, you may want to continue to use MakeProcInstance() and export all callbacks. This macro does nothing on Windows NT.

In short,

	On Windows	Win32
Callbacks	Export or use MakeProcInstance	Use address of fn

GetProcAddress	Must export	Must export
Static linking	Must export	Must export

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseDll

Exporting Data from a DLL or an Application

PSS ID Number: Q90530

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

It is possible for a Win32-based application to be able to address DLL global variables directly by name from within the executable. This is done by exporting global data names in a way that is similar to the way you export a DLL function name. Use the following steps to declare and utilize exported global data.

1. Define the global variables in the DLL code. For example:

```
int i = 1;
int *j = 2;
char *sz = "WBGLMCMTP";
```

2. Export the variables in the module-definition (DEF) file. With the 3.1 SDK linker, use of the CONSTANT keyword is required, as shown below:

```
EXPORTS
   i  CONSTANT
   j  CONSTANT
   sz CONSTANT
```

   With the 3.5 SDK linker or the Visual C++ linker, use of the DATA keyword is required, as shown below

```
EXPORTS
   i  DATA
   j  DATA
   sz DATA
```

   Otherwise, you will receive the warning

```
   warning LNK4087: CONSTANT keyword is obsolete; use DATA
```

   Alternately, with Visual C++, you can export the variables with:

```
   _declspec( dllexport ) int i;
   _declspec( dllexport ) int *j;
   _declspec( dllexport ) char *sz;
```

3. If you are using the 3.1 SDK, declare the variables in the modules that will use them (note that they must be declared as pointers because a

pointer to the variable is exported, not the variable itself):

```
extern int *i;
extern int **j;
extern char **sz;
```

If you are using the 3.5 SDK or Visual C++ and are using DATA, declare the variables with _declspec(dllimport) to avoid having to manually perform the extra level of indirection:

```
_declspec( dllimport ) int i;
_declspec( dllimport ) int *j;
_declspec( dllimport ) char *sz;
```

4. If you did not use _declspec(dllimport) in step 3, use the values by dereferencing the pointers declared:

```
printf( "%d", *i );
printf( "%d", **j );
printf( "%s", *sz );
```

It may simplify things to use #defines instead; then the variables can be used exactly as defined in the DLL:

```
#define i *i
#define j *j
#define sz *sz

extern int i;
extern int *j;
extern char *sz;

printf( "%d", i );
printf( "%d", *j );
printf( "%s", sz );
```

MORE INFORMATION

NOTE: This technique can also be used to export a global variable from an application so that it can be used in a DLL.

REFERENCE

For more information on the use of EXPORTS and CONSTANT in the Module Definition File (DEF) file for the 3.1 SDK, see Chapter 4 of the Win32 SDK "Tools" manual.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseDll

Extending Standard Windows Controls Through Superclassing

PSS ID Number: Q76947

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

A Windows-based application can extend the behavior of a standard Windows control by using the technique of superclassing. An application can superclass a standard Windows control by retrieving its window class information, modifying the fields of the WNDCLASS structure, and registering a new class. For example, to associate status information with each button control in an application, the buttons can be superclassed to provide a number of window extra bytes.

This article describes a technique to access the WNDCLASS structure associated with the standard "button" class.

MORE INFORMATION

The following five steps are necessary to register a new class that uses some information from the standard windows "button" class:

1. Call GetClassInfo() to fill the WNDCLASS structure.

2. Save the cbWndExtra value in a global variable.

3. Add the desired number of bytes to the existing cbWndExtra value.

4. Change the lpszClassName field.

5. Call RegisterClass() to register the new class.

The first step will fill the WNDCLASS structure with the data that was used when the class was originally registered. In this example, the second step is necessary so that when the "new" extra bytes are accessed, the original extra bytes are not destroyed. Please note that it is NOT safe to assume that the original cbWndExtra value was zero. When accessing the "new" extra bytes, it is necessary to use the original value of cbWndExtra as the base for any new data stored in the extra bytes. The third step allocates the new extra bytes. The fourth step specifies the new name of the class to be registered, and the final step actually registers the new class.

Any new class created in this manner MUST have a unique class name.

Typically, this name would be similar but not identical to the original class. For example, to superclass a button, an appropriate class name might be "superbutton." There is no conflict with class names used by other applications as long as the CS_GLOBALCLASS class style is not specified. The standard Windows "button" class remains unchanged and can still be used by the application as normal. In addition, once a new class has been registered, any number of controls can be created and destroyed with no extra coding effort. The superclass is simply another class in the pool of classes that can be used when creating a window.

The sample code below demonstrates this procedure:

```
BOOL DefineSuperButtonClass(void)
{
#define MYEXTRABYTES 8

   HWND     hButton;
   WNDCLASS wc;
   static char pszClassName[] = "superbutton";

   GetClassInfo(NULL, "button", (LPWNDCLASS)&wc);

   iStdButtonWndExtra = wc.cbWndExtra;   // Save this in a global

   wc.cbWndExtra += MYEXTRABYTES;

   wc.lpszClassName= pszClassName;

   return(RegisterClass((LPWNDCLASS)&wc));
}
```

It is important to note that the lpszClassName, lpszMenuName, and hInstance fields in the WNDCLASS structure are NOT returned by the GetClassInfo() function. Please refer to page 4-153 of the "Microsoft Windows Software Development Kit Reference Volume 1" for more information. Also, each time a new class is registered, scarce system resources are used. If it is necessary to alter many different standard classes, the GetProp(), SetProp(), and RemoveProp() functions should be used as an alternative approach to associating extra information with standard Windows controls.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Extracting the SID from an ACE

PSS ID Number: Q102101

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

To access the security identifier (SID) contained in an access control entry (ACE), the following syntax can be used:

```
PSID pSID;

if((((PACE_HEADER)pTempAce)->AceType) == ACCESS_ALLOWED_ACE_TYPE)
{
   pSID=(PSID)&((PACCESS_ALLOWED_ACE)pTempAce)->SidStart;
}
```

MORE INFORMATION

================

The "if" statement checks the type of ACE, which is one of the following values:

```
ACCESS_ALLOWED_ACE_TYPE
ACCESS_DENIED_ACE_TYPE
SYSTEM_AUDIT_ACE_TYPE
```

The conditional statement casts pTempAce (the pointer to the ACE) to a PACCESS_ALLOWED_ACE structure. The address of the SidStart member is then cast to a PSID and assigned to the pSID variable. pSID can now be used with any Win32 Security application programming interface (API) that takes a PSID as a parameter.

Additional reference words: 3.10 3.50

KBCategory: kbprg

KBSubcategory: BseSecurity

FADEIN: Sample App Uses Palette Animation & Identity Palettes

PSS ID Number: Q149855

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

SUMMARY

=======

A popular method used to perform a fade on an image is to animate the palette. An application displays a bitmap, and then animates the palette to all black, fading the image into blackness.

To perform the fade in the opposite direction, starting with all black and fading into color, requires some additional ingenuity. The original BitBlt() that gets the image's bits on the screen must be done without color matching. Because all of the pixels are intended to be black, if color matching were used, the pixels would be matched to the same palette entry. A subsequent palette animation would result in a single-colored rectangle.

You can take advantage of an optimization in Windows to work around this issue. When Windows is performing a BitBlt and it detects that an identity palette is being used, no color matching is performed. The source bits are moved to the destination unchanged. To perform a fade from black to color, use an identity palette to get the bits onto the screen and then fade the palette from all black to the original bitmap's desired colors.

To prepare a bitmap to use an identity palette, you must reduce the image from 256 colors to 236. An identity palette must contain the system entries in its first and last ten entries. The code in the sample below that reduces the image to 236 colors must first calculate the optimal 236 colors from a 256 color table. It then determines how close the colors are to each other, and discards colors that have close matches while keeping colors that have no close matches. Once it determines the best 236 colors, BitBlt() maps the 256 color image to 236 colors. Everything is then moved up ten to allow for the system colors in the identity palette.

This sample application in FADEIN.EXE contains code that performs the following tasks:

- Creates a DIBSection from a bitmap stored as a resource
- Chooses the optimal 236 colors from a 256 color table
- Reduces a 256 color DIBSection to 236 colors
- Creates an identity palette
- Performs palette animation to fade from black to color

MORE INFORMATION

==========

Download FADEIN.EXE, a self-extracting file, from the Microsoft
Software Library (MSL) on the following services:

You can find FADEIN.EXE, a self-extracting file, on these services:

 - Microsoft's World Wide Web site on the Internet
      On the www.microsoft.com home page, click the Support icon
      Click Knowledge Base, and select the product
      Enter kbfile FADEIN.EXE, and click GO!
      Open the article, and click the button to download the file

 - Internet (anonymous FTP)
      ftp ftp.microsoft.com
      Change to the Softlib/Mslfiles folder
      Get FADEIN.EXE

 - The Microsoft Network
      On the Edit menu, click Go To, and then click Other Location
      Type "mssupport" (without the quotation marks)
      Double-click the MS Software Library icon
      Find the appropriate product area
      Locate and Download FADEIN.EXE

 - Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download FADEIN.EXE

For additional information about downloading, please see the following
article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q119591
   TITLE   : How to Obtain Microsoft Support Files from Online Services

Additional reference words: 3.50 AnimatePalette CreateDIBSection
KBCategory: kbgraphic kbfile kbcode kbhowto
KBSubcategory: GdiPal

Fault Handling Logic Changed for Windows 95

PSS ID Number: Q141203

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95

SUMMARY

=======

Windows 95 has modified the way VxD fault handlers should handle passing the fault on to the previous handler. Using fault handler logic from a Windows 3.10 VxD may cause problems in Windows 95.

MORE INFORMATION

================

In Windows 3.1x, the following code logic might have been used by a fault handler:

```
   pPrevFaultHandler dd ?

      mov     eax, fault_number
      mov     esi, offset32 FaultHandler
      VMMCall Hook_V86_Fault
      mov     pPrevFaultHandler, esi

   BeginProc FaultHandler
      ;;;
      ;;; handler code
      ;;;

      cmp     pPrevFaultHandler, 0
      jz     @F
      jmp     pPrevFaultHandler
   @@:   ret
   EndProc FaultHandler
```

In Windows 95, this logic should be modified as follows:

```
   pPrevFaultHandler dd 0

      mov     eax, fault_number
      mov     esi, offset32 FaultHandler
      VMMCall Hook_V86_Fault
   ;   NOTE:  No "mov pPrevFaultHandler, esi" instruction
   ;   esi = 0 if this is the first fault handler
   ;   pPrevFaultHandler will *always* be nonzero.
   ;   if esi = 0, pPrevFaultHandler will be the address
   ;   of the default handler.

      ...
```

```
      mov     eax, fault_number
      mov     esi, offset32 FaultHandler
      VMMCall UnHook_V86_Fault

   BeginProc FaultHandler, HOOK_PROC, pPrevFaultHandler
      ;;;
      ;;; handler code
      ;;;

   ;   NOTE: No "cmp pPrevFaultHandler, 0" instruction
      jmp   pPrevFaultHandler
   EndProc FaultHandler
```

Additional reference words: 4.00
KBCategory: kbprg kbcode
KBSubcategory:

File Manager Passes Short Filename as Parameter

PSS ID Number: Q98575

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

When starting an application from File Manager by double-clicking a document associated with the application, if the document resides on an NTFS partition and has a long (non-8.3 form) filename, File Manager will pass the short version of the filename (also known as the MS-DOS alias or 8.3 name) to the associated application if the application is an MS-DOS or 16-bit Windows-based application. This is done for compatibility reasons; applications not aware of long filenames (16-bit Windows-based applications) can still function correctly. 32-bit Windows-based applications will be passed the long file name.

This can create confusion, however, if the application displays the name of the file the application was started with; the short name is displayed even though the long name was double-clicked.

You can avoid possible confusion by always expanding any filenames passed to an application via the command line. Do this by calling the FindFirstFile() application programming interface (API) on these filenames. FindFirstFile() will always return the file system's version of the filename in the WIN32_FIND_DATA.cFileName structure member, which the application can then use in all further references to the file without any problems.

Additional reference words: 3.10 file name 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

FILE_FLAG_WRITE_THROUGH and FILE_FLAG_NO_BUFFERING

PSS ID Number: Q99794

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The FILE_FLAG_WRITE_THROUGH flag for CreateFile() causes any writes made to that handle to be written directly to the file without being buffered. The data is cached (stored in the disk cache); however, it is still written directly to the file. This method allows a read operation on that data to satisfy the read request from cached data (if it's still there), rather than having to do a file read to get the data. The write call doesn't return until the data is written to the file. This applies to remote writes as well--the network redirector passes the FILE_FLAG_WRITE_THROUGH flag to the server so that the server knows not to satisfy the write request until the data is written to the file.

The FILE_FLAG_NO_BUFFERING takes this concept one step further and eliminates all read-ahead file buffering and disk caching as well, so that all reads are guaranteed to come from the file and not from any system buffer or disk cache. When using FILE_FLAG_NO_BUFFERING, disk reads and writes must be done on sector boundaries, and buffer addresses must be aligned on disk sector boundaries in memory.

These restrictions are necessary because the buffer that you pass to the read or write API is used directly for I/O at the device level; at that level, your buffer addresses and sector sizes must satisfy any processor and media alignment restrictions of the hardware you are running on.

MORE INFORMATION

This code fragment demonstrates how to sector-align data in a buffer and pass it to CreateFile():

```
  char buf[2 * SECTOR_SIZE - 1], *p;

  p = (char *) ((DWORD) (buf + SECTOR_SIZE - 1) & ~(SECTOR_SIZE - 1));
  h = CreateFile(argv[1], GENERIC_READ | GENERIC_WRITE,
      FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
      FILE_ATTRIBUTE_NORMAL | FILE_FLAG_NO_BUFFERING, NULL);
  WriteFile(h, p, SECTOR_SIZE, &dwWritten, NULL);
```

The pointer p is sector-aligned and points within the buffer.

If you have a situation where you want to flush all open files on the

current logical drive, this can be done by:

```
hFile = CreateFile("\\\\.\\c:", ....);
FlushFileBuffers(hFile);
```

This method causes all buffered write data for all open files on the C: partition to be flushed and written to the disk. Note that any buffering done by anything other than the system is not affected by this flush; any possible file buffering that the C Run-time is doing on files opened with C Run-time routines is unaffected.

When opening a remote file over the network, the server always caches and ignores the no buffering flag specified by the client. This is by design. The redirector and server cannot properly implement the full semantics of FILE_FLAG_NO_BUFFERING over the network. In particular, the requirement for sector-sized, sector-aligned I/O cannot be met. Therefore, when a Win32-based application asks for FILE_FLAG_NO_BUFFERING, the redirector and server treat this as a request for FILE_FLAG_WRITE_THROUGH. The file is not cached at the client, writes go directly to the server and to the disk on the server, and the read/write sizes on the network are exactly what the application asks for. However, the file is cached on the server.

Not caching the client can have a different effect, depending on the type of I/O. You eliminate the cache hits or read ahead, but you also may reduce the size of transmits and receives. In general, for sequential I/O, it is a good idea to cache on the client. For small, random access I/O, it is often best not to cache.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseFileio

FILE_READ_EA and FILE_WRITE_EA Specific Types

PSS ID Number: Q102104

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

The FILE_READ_EA and FILE_WRITE_EA specific types provide access to read and write a file's extended attributes. Specific access types are represented as bits in the access mask and are specific to the object type associated with the mask.

Please note that these specific types are used in the definition of constants such as FILE_GENERIC_READ, and are not intended to be generally used when specifying access (generic access types are much more appropriate).

Additional reference words: 3.10
KBCategory: kbprg
KBSubcategory: BseFileio

Filenames Ending with Space or Period Not Supported

PSS ID Number: Q115827

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

CreateFile() removes trailing spaces and periods from file and directory names. This is done for compatibility with the FAT and HPFS file systems.

Problems can arise when a Macintosh client creates a file on a Windows NT server. The code to remove trailing spaces and periods is not carried out and the Macintosh user gets the correctly punctuated filename. The Win32 APIs FindFirstFile() and FindNextFile() return a filename that ends in a space or in a period; however, there is no way to create or open the file using the Win32 API.

Applications such as File Manager and Backup check to see whether the filename ends with a space or period. If the filename does end in a space or a period, then File Manager and Backup use the alternative name found in WIN32_FIND_DATA.cAlternateFileName to create and open the file. Therefore, the full filename is lost.

Additional reference words: 3.10 3.50 4.00 95 winfile ntbackup
KBCategory: kbprg
KBSubcategory: BseFileio

FileTimeToLocalFileTime() Adjusts for Daylight Saving Time

PSS ID Number: Q128126

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SUMMARY

Under NTFS, the API GetFileTime() returns the create time, last access time, and last write time for the specified file. The times returned in the FILETIME structures are in Universal Coordinated Time (UTC). This is also the time that NTFS uses. You can use FileTimeToLocalFileTime() to convert a file time to a local time. However, if you automatically adjust for Daylight Saving Time, FileTimeToLocalFileTime() will adjust for Daylight Saving Time based on whether the current date should be adjusted for Daylight Saving Time, not based on whether the date represented by the FILETIME structure should be adjusted.

The behavior in this situation is different under FAT, but may be changed to match the behavior under NTFS in a future version of Windows NT.

MORE INFORMATION

The result of this behavior, which is by design, is that reported file times under NTFS may change with the start and end of Daylight Saving Time. For example, suppose that the file TEST.C has a last write FILETIME representing Jan 1, 1995 9:00pm (UTC), it is not Daylight Saving Time, and you are in the Pacific time zone. Both the DIR command and the following sample code report the file time as 1:00pm (LocalTime = UTC - 8).

Sample Code 1

```
   #include <windows.h>

   void main()
   {
      HANDLE hFile;
      FILETIME ftCreate, ftLastAccess, ftLastWrite, ftLocal;
      SYSTEMTIME st;

      char buf[80];

   // Open the file.

      hFile = CreateFile( "test.c",
                          GENERIC_READ,
                          0,
```

```
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL );

   // Get the file time (in UTC) and convert to local time.

      GetFileTime( hFile, &ftCreate, &ftLastAccess, &ftLastWrite );
      FileTimeToLocalFileTime( &ftLastWrite, &ftLocal );

   // Display the time, as a test.

      FileTimeToSystemTime( &ftLocal, &st );
      GetTimeFormat( LOCALE_USER_DEFAULT, 0, &st, NULL, buf, sizeof(buf) );
      MessageBox( NULL, buf, " FILE TIME", MB_OK );
   }
```

Now, set the date to 7/1/95 and enable Automatically Adjust for Daylight Saving Time. The DIR command and the sample code above will report the file time as 2:00pm, because FileTimeToLocalFileTime() has adjusted for Daylight Saving Time (LocalTime = UTC - 7).

The following sample code will correctly report the file time of TEST.C with the date set to 7/1/95 under NTFS. The FILETIME structure is converted to a SYSTEMTIME structure with FileTimeToSystemTime(). Then the time is converted using SystemTimeToTzSpecificLocalTime(). If you need to convert back to a FILETIME structure, use SystemTimeToFileTime() after the conversion to local time.

Sample Code 2

```
   #include <windows.h>

   void main()
   {
      HANDLE hFile;
      FILETIME ftCreate, ftLastAccess, ftLastWrite;
      SYSTEMTIME stUTC, st;
      char buf[80];

   // Open the file.

      hFile = CreateFile( "test.c",
                        GENERIC_READ,
                        0,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL,
                        NULL );

   // Get the file time (in UTC) and convert to local time.

      GetFileTime( hFile, &ftCreate, &ftLastAccess, &ftLastWrite );
      FileTimeToSystemTime( &ftLastWrite, &stUTC );
```

```
      SystemTimeToTzSpecificLocalTime( NULL, &stUTC, &st);

   // Display the time, as a test.

      GetTimeFormat( LOCALE_USER_DEFAULT, 0, &st, NULL, buf, sizeof(buf) );
      MessageBox( NULL, buf, "FILE TIME", MB_OK );
   }
```

The FAT file system stores local time, not UTC. GetFileTime() gets cached UTC times from FAT. In this sample, the time stored is 1pm and the cached time is 9pm. When it becomes Daylight Saving Time, sample codes 1 and 2 will demonstrate the same behavior that they do under NTFS, because 9pm is still used. However, when you restart the machine, the new cached time will be 8pm (UTC = LocalTime + 7). The call to FileTimeToLocalFileTime() cancels the adjustment made by GetFileTime() (LocalTime = UTC - 7). Therefore, sample code 1 will report the correct time under FAT, but sample code 2 will not.

On the other hand, FindFirstFile() on FAT always reads the time from the file (stored as local time) and converts it into UTC, adjusting for DST based on the current date. So if FindFirstFile() is called, the date is changed to a different DST season, and then FindFirstFile() is called again, the UTC returned by the two calls will be different.

Additional reference words: 3.50
KBCategory: kbprg kbcode
KBSubcategory: BseMisc

Filtering All Paging File I/O Under Windows 95

PSS ID Number: Q141204

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95

SUMMARY

A file system API hook installed by a VxD by calling IFSMGR_InstallFileSystemAPIHook will not see initial file I/O requests for the paging file unless it installs the hook during Device_Init.

MORE INFORMATION

Some file system API hooker VxDs need to see all file I/O, including all the initial requests for the paging file used by Windows. The FS_ReadFile and FS_WriteFile requests will be sent for the paging file with the R0_SWAPPER_CALL flag set for paging file I/O. The paging file is opened at the Init_Complete system control message sent to VxDs. If the VxD installs the hook during Device_Init, it will see all paging file I/O requests.

Additional reference words: 4.00
KBCategory: kbprg
KBSubcategory:

First and Second Chance Exception Handling

PSS ID Number: Q105675

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

Structured exception handling (SEH) takes a little getting used to, particularly when debugging. It is common practice to use SEH as a signaling mechanism. Some application programming interfaces (APIs) register an exception handler in anticipation of a failure condition that is expected to occur in a lower layer. When the exception occurs, the handler may correct or ignore the condition rather than allowing a failure to propagate up through intervening layers. This is very handy in complex environments such as networks where partial failures are expected and it is not desirable to fail an entire operation simply because one of several optional parts failed. In this case, the exception can be handled so that the application is not aware that an exception has occurred.

However, if the application is being debugged, it is important to realize that the debugger will see all exceptions before the program does. This is the distinction between the first and second chance exception. The debugger gets the "first chance," hence the name. If the debugger continues the exception unhandled, the program will see the exception as usual. If the program does not handle the exception, the debugger will see it again (the "second chance"). In this latter case, the program normally would have crashed had the debugger not been present.

If you do not want to see the first chance exception in the debugger, then disable the feature. Otherwise, during execution, when the debugger gets the first chance, continue the exception unhandled and allow the program to handle the exception as usual. Check the documentation for the debugger that you are using for descriptions of the commands to be used.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseExcept

FIX: Corruption of the Perflib Registry Values

PSS ID Number: Q128404

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SYMPTOMS
========

The Performance Monitoring tool can be affected by a corruption problem in the Perflib key. The following symptoms are evidence of this corruption problem:

- The explain text is not displayed.

  -or-

- The objects and counters are not displayed.

  -or-

- The explain text, objects, and counters are not displayed.

CAUSE
=====

When an .INI file, which contains an extra language entry not associated with any specific objects and counters, is passed as a parameter to LODCTR.EXE, the values of 'Last Counter' and 'Last Help' are set to zero. This in turn causes the Performance Monitoring tool to fail.

The following is an example of a problem-causing .INI file:

```
[info]
drivername=SampPerf
symbolfile=sampperf.h

[languages]
009=English
011=OtherLanguage  <-- problem area

[text]
SampObj_009_Name=SampPerf
SampObj_009_Help=A sample performance object.

Count_1_009_Name=Tests/sec
Count_1_009_Help=The number of tests completed.
```

A further complication arises when a user attempts to rectify the situation by executing UNLODCTR.EXE. At this point, because Last Counter and Last

Help are set to zero, UNLODCTR.EXE only resets '\009\HELP' and '\009\COUNTER' to NULL.

RESOLUTION

To resolve this corruption problem, you must restore the Registry to its former state by using one of the following five methods:

- Backup and restore the local copy of the Registry by using Windows NT Backup.

  -or-

- Copy and restore the data located in the %WINNT%\SYSTEM32\CONFIG directory. This can be done only if Windows NT was installed on a FAT partition.

  -or-

- Save and restore the SOFTWARE registry hive.

  -or-

- Save and restore the Registry values First Counter, First Help, Last Counter, and Last Help.

  -or-

- Save and restore the Registry by using REGBACK.EXE and REGREST.EXE. Both programs are available with the Windows NT Resource Kit (RESKIT).

If the user ran UNLODCTR.EXE to attempt to fix the problem, you will also need to restore \009\HELP and \009\COUNTER, which you can do by simply rebooting the computer running Windows NT.

WARNING: The Registry is a vital part of Windows NT; improper modification of its keys and values can cause Windows NT to malfunction.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Windows NT 3.51.

Additional reference words: 3.50
KBCategory: kbprg kbfixlist kbbuglist
KBSubcategory: BseMisc

FIX: Error C2664 'AVIFileCreateStream' from Visual C++ 2.0/2.1

PSS ID Number: Q139825

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, 4.0
- Microsoft Visual C++, 32-bit Edition, versions 2.0 and 2.1

SYMPTOMS

The following error is generated when you compile an application under Visual C++ version 2.0 or 2.1 that calls AVIFileCreateStream() and does not use UNICODE:

```
error C2664: 'AVIFileCreateStream' : cannot convert parameter 3 from
             'struct _AVISTREAMINFOA*' to 'struct _AVISTREAMINFOW*'
```

CAUSE

The Vfw.h header file from Visual C++ version 2.0 or 2.1 has the following declaration for the AVIFileCreateStream() function:

```
STDAPI AVIFileCreateStream(PAVIFILE pfile,
           PAVISTREAM FAR *ppavi,
                          AVISTREAMINFOW FAR *psi);
```

Note the wide version of the structure in the third parameter: AVISTREAMINFOW. There is no ANSI declaration for this function in vfw.h. The header file is incorrect. It should list an ANSI version of the function that takes an ANSI AVISTREAMINFO in addition to the UNICODE version.

STATUS

Microsoft has confirmed this to be a bug in Visual C++ versions 2.0 and 2.1. This bug was corrected in Visual C++ version 2.2. The Vfw.h header file from that product declares an AVIFileCreateStreamA() function in addition to the W variety, AVIFileCreateStreamW().

Additional reference words: 3.50 2.10 2.00 4.00 Windows 95
KBCategory: kbmm kbprg kbfixlist kbbuglist
KBSubcategory: MMVideo

FIX: Error Messages or Calls to Winhelp() 4.0 Fail with Win32s

PSS ID Number: Q147867

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Application Programming Interface (API) included with Microsoft Win32s versions 1.3 and 1.3a

---

SYMPTOMS

You may encounter the following problems with the Winhelp version 4.0 program included with Win32s versions 1.30 and 1.30a if Windows NT is installed in the same directory as Win32s.

- Calls to WinHelp() from within a 32-bit application fail.

- One of the following error messages may occur:

  An error exists in your Help file. Contact your application vendor for an updated Help file (1053)

  -or-

  Error: This app uses CTL3D32.dll which is not the correct version. This version is for Windows NT Systems only.

CAUSE

These problems are caused by the way that Win32s 1.30 and 1.30a searches the path for DLL files, specifically, the SYSTEM32 directory is searched before SYSTEM. Because of this, Winhelp finds component files that are designed for Windows NT and will not run under Win32s.

RESOLUTION

This problem has been corrected in Win32s version 1.30c. For information on how to obtain the latest version of Win32s, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q122235
TITLE : Microsoft Win32s Upgrade

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Win32s version 1.30c.

Additional reference words: 3.10 1.30 1.30a
KBCategory: kbprg kbfixlist kbbuglist
KBSubcategory: W32s

FIX: Floating Point Exception Incorrect Under Win32s

PSS ID Number: Q150405

The information in this article applies to:

- Microsoft Win32s version 1.3

SYMPTOMS
========

There are some inconsistencies in floating point exception handling under Win32s and Windows NT platforms. For example, the following code causes a underflow on both platforms as expected and the exception is masked:

```
d1 = pow(1000000, -53);
```

However, the following code:

```
d2 = 1/d1;
```

causes this behavior:

Windows NT 3.51	-	Floating point Overflow.
Windows 3.1/Win32s	-	Denormal operand error

The overflow exception under Windows NT is correct, but the result under Win32s is not.

RESOLUTION
==========

If you examine the exception record, you can see that the denormal operand status bit is set but the exception is masked in the control word. On the other hand, the overflow exception status bit is also set and the exception is unmasked. Even though the exception indication is incorrect, you can get the correct exception by examining the status and control words appropriately.

STATUS
======

Microsoft has confirmed this to be a bug in Win32s version 1.30. Win32s incorrectly sets the floating point exception in the above case to a Denormal operand error instead of an Overflow exception. This problem has been corrected in Win32s version 1.30c.

Additional reference words: 1.30 win32s fp error
KBCategory: kbprg kbbuglist kbfixlist
KBSubcategory: w32s

FIX: High CPU Usage When SNMP Retrieves Performance Counters

PSS ID Number: Q130563

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SYMPTOMS

When an SNMP extension agent tries to retrieve performance counters, CPU utilization jumps to 100 percent.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Windows NT version 3.51.

MORE INFORMATION

If an SNMP extension agent tries to retrieve performance counters as in the following example, the CPU utilization is observed to jump to 100 percent and stay there until the SNMP service is stopped.

```
RegOpenKeyEx(...);
.
.
RegQueryValueEx(HKEY_PERFORMANCE_DATA,
...)
```

To see this, use the Performance Monitor tool in the Administrative Tools program group.

The same problem can occur when an application is trying to retrieve TCP/IP performance counters and the Internet MIB II agent is being used to retrieve the counters.

REFERENCES

For more information, please see the Windows NT 3.5 Resource Kit Vol IV.

Additional reference words: 3.50
KBCategory: kbnetwork kbfixlist kbbuglist
KBSubcategory: NtwkSnmp

FIX: MCIWndCreate Does Not Allow Filename

PSS ID Number: Q132452

The information in this article applies to:

- Microsoft Windows NT version 3.5

SYMPTOMS

If a filename is specified when MCIWndCreate is called (that is, the last parameter is not NULL), an MCI Error occurs giving this message:

> Cannot find the specified file. Make sure the path and file name are correct.

RESOLUTION

This bug was fixed in Windows NT version 3.51. The workaround is to pass NULL as the filename when calling MCIWndCreate. Then call MCIWndOpen to open the file.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Windows NT version 3.51.

Additional reference words: 3.50
KBCategory: kbprg kbbuglist kbfixlist
KBSubcategory: MMMisc MCIWnd

FIX: MFC-Based App Cannot Run Under International Win32s

PSS ID Number: Q125457

The information in this article applies to:

 - Microsoft Win32s version 1.2

SYMPTOMS
========

When activating a Microsoft Foundations Classes (MFC)-based application built with Visual C++ version 2.0 under Win32s version 1.2, if the language setting in Windows or Windows for Workgroups Control panel is set to anything other than English, German, or French, the following error occurs:

   Win32s - Error: Initialization of a dynamic link library failed.
   The process is terminating abnormally.

This error has also been reported with non-MFC-based applications as well.

CAUSE
=====

CRT startup code causes an unhandled exception in the Win32s DLL (W32SCOMB.DLL).

RESOLUTION
==========

For an international version of Windows 3.1 or Windows for Workgroups, if the country and language settings are changed to United States and English(American), 32-bit MFC-based applications are able to run.

STATUS
======

Microsoft has confirmed this to be a bug in Win32s version 1.2. This problem has been corrected in Win32s version 1.25.

Additional reference words: 1.20 w32scomb
KBCategory: kbinterop kbfixlist kbbuglist
KBSubcategory: W32s WIntlDev

FIX: MIDL 2.0 Does Not Handle VAX Floating Point Numbers

PSS ID Number: Q129064

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SYMPTOMS
========

RPC applications (client and server) built using MIDL 2.0 (shipped with Windows NT version 3.5) may get spurious characters or report memory violations when dealing with VAX floating point numbers.

RESOLUTION
==========

There are two possible resolutions to this problem:

- Use MIDL 1.0 (shipped with Windows NT version 3.1) to build RPC client and server applications.

  -or-

- On the VAX side, use a compiler switch to make sure the application represents its floating point numbers in IEEE format.

STATUS
======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Windows NT version 3.51.

MORE INFORMATION
================

Note that Microsoft RPC is binary compatible with DCE RPC. There are four types of representations for floating point numbers: IEEE, VAX, IBM, and CRAY. An application built using MIDL 1.0 can recognize IEEE and VAX representations but not IBM and CRAY. Applications built using MIDL 2.0 can recognize only IEEE representaion as mentioned above. Both, MIDL 1.0 and 2.0, use IEEE representations to send data.

Additional reference words: 3.50
KBCategory: kbnetwork kbfixlist kbbuglist
KBSubcategory: NtwkRpc

FIX: OLE Libraries Fix List

PSS ID Number: Q134799

The information in this article applies to:

- Microsoft OLE Libraries for Windows and Win32s, versions 2.02, 2.03
- Microsoft Win32s, versions 1.2, 1.25a, and 1.3

Fixes in OLE 2.02 (Win32s 1.2)

The following is a list of the known bugs that were fixed in OLE 2.02, which is distributed with Win32s version 1.2.

- Changing the source of a link when the object is iconic does not update the new link name.

- GetClassfile fails if the offset is greater than 32K.

- GetClassfile returns NO_ERROR when the clsid string in the registry contains illegal characters.

- The IStream returned from CreateStreamOnHGlobal does not support QueryInterface correctly.

- GetClassfile fails if the offset is negative.

- For composite monikers, IMoniker::CommonPrefixWith returns the wrong hresult if both monikers have a common prefix. For item monikers, IMoniker::CommonPrefixWith doesn't handle the case that the other moniker is not an item moniker.

- CreateGenericComposite fails when handling an item moniker with an anti moniker.

- For item monikers, IMoniker::ParseDisplayName causes a general protection (GP) fault when pmkToLeft is NULL.

- For anti monikers, IMoniker::RelativePathTo returns an interface pointer without first calling AddRef and returns a wrong hresult.

- The ILockBytes returned from CreateILockBytesOnHGlobal does not support QueryInterface correctly.

- ICLassFactory::LockServer returns the wrong hresult.

- OLE time APIs do not validate input pointers.

- For composite monikers, calling IMoniker::RelativePathTo leaks memory.

- OleLoadFromStream(pstrm, IID_IUnknows, &pmk) always returns E_NOINTERFACEW (0x80004002).

- When registering a new pointer to unknown with an existing key, IBindCtx::RegisterObjectParam does not release the old pointer, resulting in a memory leak.

- IBindCtx::RevokeObjectBound fails with MK_E_NOTBOUND even though the object is in the list.

- IPersistFile::IsDirty fails.

- There is no marshalling code for IPersistStream.

- For pointer monikers, IMoniker::IsRunning is not implemented correctly.

- Drawing a metafile causes a leak of many palettes.

- OleMetafilePictFromIconAndLabel mishandles long labels.

- OleMetafilePictFromIconAndLabel tries to write to a label.

Fixes in OLE 2.03 (Win32s 1.25)

The following is a list of the known bugs that were fixed in OLE 2.03, which is distributed with Win32s version 1.25.

- IMalloc::DidAlloc does not return 1 for memory it allocates.

- There is no marshalling code for IEnumUnknown, IMoniker, and IEnumMoniker.

- MkParseDisplayName silently fails if the server is already running.

- Wrong cbSize in OBJECTDESCRIPTOR.

- GPF occurs when marshalling a MSG structure in 16-32 interop.

- Bug in remoting of OA interfaces (32-bit only).

- Bug in second activation of the olespy tool. It may not run.

- OleConvertOLESTREAMToIStorage always saves CLSID_NULL instead of the real clsid.

- GPF marshalling ILockBytes::Stat, IStorage::Stat, and IStream::Stat.

- GPF with 16-32 interop when the 32-bit application installs a hook.

Fixes in OLE 2.03 (Win32s 1.25a)

The following is a list of the known bugs that were fixed in OLE 2.03, which is distributed with Win32s version 1.25a.

- Can't unload an Excel Add-in.

- Cross-bitness QueryInterface does not return the correct error code.

- Japanese NEC PCs may crash when calling CoCreateGuid.

- Failure to call a 32-bit OA server if the calling chain is too long. Therefore, you can not do the following: OA controller -> OA serverA -> OA serverB -> OA serverC.

Fixes in 32-bit OLE (Win32s 1.3)

The following is a list of the known bugs that were fixed in the 32-bit OLE which is distributed with Win32s version 1.3.

- General protection (GP) fault when a 32-bit container activates a 16-bit server more than once and uses both OA intefaces and core OLE interfaces.

- When a call to OleGetClipboard fails, the clipboard become corrupted and other applications can not use it.

- A 16-bit container copies data to the clipboard and a 32-bit server uses the Paste Special dialog. The string from the object descriptor is corrupted and a general protection (GP) fault may occur.

Additional reference words: 1.20 1.30
KBCategory: kbprg kbfixlist kbbuglist
KBSubcategory: LeTwoMisc W32s

FIX: Owner Drawn Items on the Menu Bar Hang Windows NT

PSS ID Number: Q126720

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SYMPTOMS
========

Windows NT supports owner drawn items on the menu bar. However, Windows NT may stop responding (hang) if multiple windows of same class are created with the owner drawn menu bar.

STATUS
======

Microsoft has confirmed this to be a bug in Windows NT version 3.50. This problems was corrected in Windows NT version 3.51.

MORE INFORMATION
================

Steps to Reproduce Problem
----

1. Assign a menu to a window class via RegisterClass().

2. Modify the menu bar to be owner draw in WM_CREATE.

3. Create multiple windows of the same window class.

Windows NT may stop responding, in which case, you must turn off the computer.

Additional reference words: 3.50
KBCategory: kbui kbfixlist kbbuglist
KBSubcategory: UsrMen UsrOwn

FIX: Sample File Fixes OpenGL's DIB-Resizing Bug

PSS ID Number: Q145888

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Microsoft Windows 95

SYMPTOMS

On the Windows 95 version of OpenGL, if you use OpenGL to render to a bitmap by using a memory DC and a new (larger) bitmap is selected into the memory DC, then the new bitmap dimensions are not recognized and drawing is limited to the old bitmap size.

RESOLUTION

Download the fix (OGLFIX.EXE), and then read the Readme.txt file included before installing the fix onto your system.

You can find OGLFIX.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  - On the www.microsoft.com home page, click the Support icon
  - Click Knowledge Base, and select the product
  - Enter kbfile OGLFIX.EXE, and click GO!
  - Open the article, and click the button to download the file

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib/Mslfiles folder
  - Get OGLFIX.EXE

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type "mssupport" (without the quotation marks)
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Locate and Download OGLFIX.EXE

- Microsoft Download Service (MSDL)
  - Dial (206) 936-6735 to connect to MSDL
  - Download OGLFIX.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

STATUS

======

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in OpenGL32.dll which is contained in OGLFIX.EXE.

Additional reference words: 4.00 OpenGL PFD_DRAW_TO_BITMAP DIB resize
KBCategory: kbgraphic kbfixlist kbbuglist kbfile
KBSubcategory: GdiOpenGL

FIX: Setsockopt() for Winsock over Appletalk Returns Error

PSS ID Number: Q129062

The information in this article applies to

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SYMPTOMS

After you open a socket of type SOCK_STREAM by using the ATPROTO_ADSP protocol, and bind to a dynamic socket, the setsockopt() function fails under these conditions:

- The setsockopt() function uses the zone name returned by getsockopt() (also found by looking at the control panel network entry) with the SO_LOOKUP_MYZONE option.

- The zone name is supplied to setsockopt() with the SO_REGISTER_NAME option.

Error code 10022 (WSAINVAL :Invalid Argument) is returned on calling GetLastError().

RESOLUTION

Instead of passing the string returned by getsockopt() for the zone name, use the character "*" for the ZoneName member of the WSH_REGISTER_NAME struct. For example, use this:

```
WSH_REGISTER_NAME regName;
........
strcpy( regName.ZoneName, "*");
```

instead of this:

```
strcpy( regName.ZoneName, "BLDG/1");
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Windows NT 3.51.

REFERENCES

Windows Sockets for Appletalk (SFMWSHAT.WRI version 1.2).

Additional reference words: 3.50
KBCategory: kbnetwork kbfixlist kbbuglist

KBSubcategory: NtwkWinsock

FIX: SNMP Sample Generates an Application Error

PSS ID Number: Q124961

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SYMPTOMS

The Win32 SDK for Windows NT 3.5 contains a SNMP sample called SNMPUTIL (\MSTOOLS\SAMPLES\WIN32\SNMP\SNMPUTIL). Building this sample generates an executable file that causes an access violation. The message box displayed resembles the following:

```
                    snmputil.exe - Application Error
   ------------------------------------------------------------
   The instruction at <address> referenced memory at <address>.  The
   memory could not be "read".
```

CAUSE

One part of the SNMP run times uses the C run-time routines provided by CRTDLL.DLL. The other part of the SNMP run times uses the C run-time routines that the application uses. Because many people are using Visual C++ to build the samples, the run time routines used are in MSVCRT20.DLL. The access violation occurs when memory allocated using one run time version are freed using a different copy of the run time library.

RESOLUTION

One possible solution is to build the sample application using the CRTDLL.DLL version of the run time library. If you are using the makefile supplied with the Win32 SDK, one way to accomplish this is to:

1. Copy NTWIN32.MAK from your include directory to the working directory for the sample.
2. Change all references to MSVCRT.LIB in NTWIN32.MAK to CRTDLL.LIB.

If you are using a Visual C++ make file, you can:

1. Select the "Ignore All Default Libraries" check box in the Linker Project Settings property page.
2. Add CRTDLL.LIB to the "Object/Library Modules" section in the Linker Project Settings property page.

NOTE: CRTDLL.LIB does not ship with Visual C++. You can get the CRTDLL.LIB file from the Win32 SDK in the \MSTOOLS\LIB\CRT\<PLATFORM> directory, where <PLATFORM> would be replaced with the type of machine you are using (I386, alpha, and so forth).

If you are using MFC 3.0, you will not be able to use CRTDLL.DLL because MFC 3.0 uses some functions that are not supplied in CRTDLL.DLL.

STATUS

Microsoft has confirmed that this is a problem in the Microsoft products listed at the beginning of this article. This problem was corrected in the Win32 SDK version 3.51.

Additional reference words: 3.50
KBCategory: kbnetwork kbnetwork kbfixlist kbbuglist
KBSubcategory: NtwkSnmp

FIX: SVCGUID.H Has Wrong UDP Port for SNMP Traps

PSS ID Number: Q129061

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SYMPTOMS

The file SVCGUID.H used by the Registration and Name Resolution (RnR) APIs has the UDP trap port for SNMP set to 167:

```
   #define SVCID_SNMP_TRAP_UDP                   SVCID_UDP(167)
```

RESOLUTION

The correct port number for SNMP traps is 162. However, please note that this entry is not looked at by the SNMP service for Windows NT. Hence, it cannot cause traps to be sent to the wrong port. SVCGUID.H is used only by Registration and Name Resolution (RnR) APIs.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This problem was corrected in Windows NT 3.51.

Additional reference words: 3.50
KBCategory: kbnetwork kbfixlist kbbuglist
KBSubcategory: NtwkWinsock

FIX: Win32s 1.25a Fix List

PSS ID Number: Q130139

The information in this article applies to:

- Microsoft Win32s, version 1.25a

The following is a list of the known bugs in Win32s version 1.2 that were fixed in Win32s version 1.25.

- GlobalAlloc(GMEM_FIXED) from 32-bit .EXE locks memory pages. It is more efficient to use GlobalAlloc(GMEM_MOVEABLE) and call GlobalFix() if necessary.

- WINMM16.DLL has no version information.

- CreateFileMapping() with SEC_NOCOMMIT returns ERROR_INVALID_PARAMETER.

- PolyPolygon() does not close the polygons.

- OpenFile() only searches the current directory when only a filename is given, not the application directory, the system directory, the windows directory, or the directories listed on the path.

- GetFileInformationByHandle() doesn't return the correct file attribute.

- GlobalUnlock() sets an error of ERROR_INVALID_PARAMETER.

- lstrcmp()/lstrcmpi() do not use the collate table correctly.

- FreeLibrary() in DLL_PROCESS_DETACH crashes the system.

- FindResource(), LoadResource(), GetProcAddress(), GetModuleFileName(), EnumResourceNames(), and other APIs, fail with a NULL hInstance.

- sndPlaySound() with SND_ASYNC | SND_MEMORY may cause a crash or may work poorly.

- Thread Local Storage (TLS) data is not initialized to 0 in TlsAlloc().

- The pointer received in the lParam of WM_INITDIALOG in the common dialog hook function becomes invalid in the following messages if the pointer is "remembered" in a static variable.

- Stubbed API GetFileAttributesW() does not return -1 on error.

- Code page CP_MACCP not supported.

- Invalid LCIDs are not recognized.

- CreateFile() fails to open an existing file in share mode.

- GetLocaleInfo() for locale returns system defaults from WIN.INI.

- GetVolumeInformation() fails with ERROR_INVALID_NAME for volumes without a label.

- VirtualProtect() may miss the last page in an address range.

- GetLocaleInfo() returns incorrect information for most non-US locales.

- ANSI/OEM conversions always use code page 437.

- GetProcAddress() for printer driver APIs is case sensitive.

- The LanMan APIs are unsupported, but they return 0, which indicates that the API was successful. They should return NERR_InvalidAPI (2142).

- CreateFileW() returns 0 instead of -1 (HFILE_ERROR).

- lstrcpyn() copies n bytes from source to destination, then appends a NULL terminator, instead of copying n-1 bytes and appending the NULL terminator.

- GetDriveType() doesn't report detecting a CD-ROM or a RAM DISK.

- CRTDLL calls TlsFree() upon each process detach, not just the last.

- If a DllEntryPoint calls FreeLibrary() when using universal thunks, the system can crash.

- Not all 32-bit DLLs have correct version numbers.

- GetCurrentDirectory() returns the wrong directory after calling GetOpenFileName(). The workaround is to call SetCurrentDirectory(".") right after returning from the call to GetOpenFileName().

- RegEnumValue() and other Registry functions return ERROR_SUCCESS even though they are not implemented. Win32s implements only the registry functions supported by Windows.

- AreFileApisANSI()/SetFileApisToANSI()/SetFileApisToOEM() are not exported. AreFileApisANSI() should always return TRUE, SetFileApiToOEM() should always fail, and SetFileApiToANSI() should always succeed.

- SetLocaleInfoW()/SetLocateInfoA() are not implemented.

- GetScrollPos() sets the last error if the scroll position is 0.

- SetScrollPos() sets the last error if the last scroll position is 0.

- LoadString() leaks memory if the string is a null string.

- GetFileVersionInfoSize() fails if the resource section is small and close to the end of the file.

- MoveFile() doesn't call SetLastError() on failure or sets a different error than on Windows NT.

- SetCurrentDirectory() does not work on a CD-ROM drive.

- GetFileVersionInfoSize() fails if the 2nd parameter is NULL.

- WSOCK32.DLL is missing exported stubs for unimplemented APIs.

- Win32s fails to load 64x64 monochrome (black and white) icons.

- CreateFile() fails when called with a filename with an international character.

- GetCurrentDirectory() returns an OEM string.

- PrintDlg() causes GP fault if hDevMode!=NULL and another printer is selected that uses a larger DevMode buffer.

- Unicode resources are not properly converted to 8-bit characters.

- CreateDC() returns an incorrect DEVMODE. This can cause a variety of symptoms, like the inability to do a Landscape Print Preview from an MFC application or the displayed paper width and height not changing, even when you change the paper size.

- OpenFile() fails on filenames with OEM characters in the name.

- WNetCancelConnection() should be supported similar to NetAddConnection().

- GetDriveType() fails on a Stacker 3.1 drive.

- SetCurrentDirectory() fails on Novell client machines.

- GetProp() returns 0 in the second instance of an app in certain cases.

- fopen(fn, "w") fails on second call.

- TLS indices allocated by a module are released when that module is freed.

- CreateWindow() handles STARTUPINFO incorrectly if the application starts minimized.

- CreateFile() creates files with incorrect attributes.

- Resource sections are now read/write to emulate the behavior of Windows NT and Windows 95.

- Removed the 128K stack limitation.

- CompareStringW() sometimes uses incorrect locale, primarily Swedish and other Scandinavian locales.

- Added dummy _iob to CRTDLL for applications that reference standard handles.

- Added support for OPENCHANNEL, CLOSECHANNEL, SEXGDIXFORM, and DOWNLOADHEADER escapes.

Additional reference words: 1.25 1.25a
KBCategory: kbprg kbfixlist kbbuglist
KBSubcategory: W32s

FIX: Win32s 1.3 Fix List

PSS ID Number: Q133027

The information in this article applies to:

- Microsoft Win32s version 1.3

The following is a list of the known bugs in Win32s version 1.25a that were fixed in Win32s version 1.3.

- DeviceCapabilities() with DC_BINS and DC_PAPERS are thunked incorrectly. The array should be left as is.

- SearchPath() and OpenFile() don't work properly with OEM chars in the filename.

- GetSystemInfo() doesn't set correct ProcessorType for the Pentium.

- FindResource() returns a non-NULL handle for a nonexistent resource.

- VirtualProtect() with anything other than PAGE_NOACCESS, PAGE_READ, OR PAGE_READWRITE yields unpredictable page protections.

- COMPAREITEMSTRUCT, DELETEITEMSTRUCT, DRAWITEMSTRUCT, AND MEASUREITEMSTRUCT incorrectly sign-extend fields.

- GetWindowTextLength() and GetWindowText() incorrectly sign-extend the return value.

- MoveFile() fails on Windows for Workgroups when the source is remote and the destination is local.

- PrintDlg() GP faults if the PRINTDLG structure has an illegal size (size other than 42).

- Loading resources may cause memory and selector leakage. The leakage occurs when the resource is loaded from a DLL.

- FP context is corrupted when an FP exception occurs while using the FP emulator.

- If a window class defined in one DLL uses a window function in another DLL and this DLL is unloaded before the DLL that defines the window class, a general protection (GP) fault occurs in WIN32S16.DLL when you terminate the process from a debugger.

- GetBitmapBits() and SetBitmapBits() return the wrong value. The return code is not converted from dx:ax to eax.

- FindResource() does not set the last error code to a proper value.

- NLS APIs fail when AnsiCP in the [NLS] section of WIN32S.INI is set to the code page of the machine (473 on U.S. machines).

- _tzset() in CRTDLL.DLL can cause a general protection (GP) fault.

In addition, there were several bugs fixed in the Windows NT FP emulator. These fixes were are included in Win32s.

REFERENCES
==========

For the list of known bugs in Win32s version 1.2 that were fixed in Win32s version 1.25a, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q130139
   TITLE   : Win32s 1.25a Fix List

Additional reference words: 1.30
KBCategory: kbprg kbfixlist kbbuglist
KBSubcategory: W32s

FIX: Win32s Version 1.3a Bugs That Were Fixed in Version 1.3c

PSS ID Number: Q147435

The information in this article applies to:

- Microsoft Win32s version 1.3a

The following is a list of known bugs in Win32s version 1.3a that were fixed in Win32s version 1.3c.

- WinHlp32 - Tab pages are not painted correctly

- Wrong Floating Point Exception reported for overflow exceptions

- DRAWITEMSTRUCT.CtlID is not sign extended for ID=-1.

- Winhlp32: In Spanish Winhlp32, character à is treated as a space character.

- GetCPInfo() returns the wrong CPINFO.LeadByte array for CP=949.

- Winhlp32 loads Windows NT's FTSRCH and COMCTL32 DLLs when Windows NT and Windows for Workgroups are installed in common directories.

- Winhlp32 behaves incorrectly when the display driver S3 v1.6 version 3.10.061 is installed.

- GetWindowTextLength() returns 0xffff instead of 0xffffffff. This may be a problem with combo boxes.

- ListView scroll bars appear sometimes when they shouldn't.

- With some display device drivers, WinHlp32's bitmaps are not drawn correctly.

- Some unimplemented shell functions do not restore the stack correctly.

- Ownerdraw Fixed ListView controls do not work properly on modeless dialogs.

- Scrolling a list view by dragging the scroll thumb does not scroll the window.

- Win32s Setup shows unexpected behavior when the System.ini file is read-only.

- The NLS sort table is freed when the process that allocated it terminates.

Additional reference words: w32s 1.30a buglist1.30a fixlist1.30c
KBCategory: kbref kbbuglist kbfixlist
KBSubcategory: w32s

FIX: Winsock Over Appletalk (DDP) Leaks Memory

PSS ID Number: Q131159

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SYMPTOMS

A memory leak occurs when you open a socket with address family AF_APPLETALK, type SOCK_DGRAM, and protocol ATPROTO_BASE + x (where x is a user-defined number except the ones that are reserved - please refer to the SDK header file ATALKWSH.H); then once a server is located to send data to, you send data to it.

Observing with PerfMon, you can see that non-paged memory use keeps on increasing. It does not decrease even after the application is stopped. If the application is allowed to continue for a long time, the results are unpredictable and it is neccessary to restart the computer.

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article. This bug was corrected in Windows NT version 3.51.

Additional reference words: 3.50 3.51
KBCategory: kbnetwork kbfixlist kbbuglist
KBSubcategory: NtwkWinsock

FixBrushOrgEx() and Brush Origins under Win32s

PSS ID Number: Q124191

The information in this article applies to:

- Microsoft Win32s versions 1.15, 1.15a, and 1.2

FixBrushOrgEx() is not implemented in the Win32 API, but it is provided for compatibility with Win32s. If called, the function does nothing, and returns FALSE.

A brush's origin relates to the origin of the window being painted. If you move a window, the brush origin needs to be updated or else newly painted patterns won't line up with the old patterns. On Windows version 3.1, the system does not automatically update the brush origin when it is selected into a device context (DC), so applications have to call SetBrushOrg(). On Windows NT, the system automatically fixes brush origins when necessary.

Win32s uses FixBrushOrgEx() to hide this difference in system behavior. On Win32s, FixBrushOrgEx() calls SetBrushOrgEx(). A Win32-based application can check the platform and call SetBrushOrgEx() only if it is Win32s, or it could simply always call FixBrushOrgEx() wherever a Windows-based application would call SetBrushOrg() for brush origin tracking.

Additional reference words: 1.20
KBCategory: kbprg
KBSubcategory: W32s

FlushViewOfFile() on Remote Files

PSS ID Number: Q95043

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

When flushing a memory-mapped file over a network, FlushViewOfFile() guarantees that the data has been written from the workstation, but not that the data resides on the remote disk.

This is because the server may be caching the data on the remote end. Therefore, FlushViewOfFile() may return before the data has been physically written to disk.

However, if the file was created via CreateFile() with the flag FILE_FLAG_WRITE_THROUGH, the remote file system will not perform lazy writes on the file, and FlushViewOfFile() will return when the actual physical write is complete.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseFileio

Format for LANGUAGE Statement in .RES Files

PSS ID Number: Q89822

The information in this article applies to:

- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

SUMMARY
=======

The syntax for the LANGUAGE statement in the resource script file is given as follows on page 289 of the Win32 SDK "Tools User's Guide" manual:

```
   LANGUAGE major, minor

   major
      Language identifier. Must be one of the constants from WINNLS.H

   minor
      Sublanguage identifier. Must be one of the constants from WINNLS.H
```

For example, suppose that you want to set the language for the resources in a file to French. For the major parameter, you would choose the following constant from the list of language identifiers

```
   #define LANG_FRENCH                 0x0c
```

and you would have the choice of any of the sublanguages that begin with SUBLANG_FRENCH in the list of sublanguage identifiers. They are:

```
   #define SUBLANG_FRENCH              0x01
   #define SUBLANG_FRENCH_BELGIAN      0x02
   #define SUBLANG_FRENCH_CANADIAN     0x03
   #define SUBLANG_FRENCH_SWISS        0x04
```

RC.EXE does not directly place these constants in the .RES file. It uses the macro MAKELANGID to turn the parameters into a WORD that corresponds to a language ID.

NOTE: The following three combinations have special meaning:

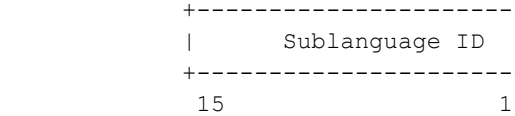
Primary language ID	Sublanguage ID	Meaning
LANG_NEUTRAL	SUBLANG_NEUTRAL	Language neutral
LANG_NEUTRAL	SUBLANG_DEFAULT	User default language
LANG_NEUTRAL	SUBLANG_SYS_DEFAULT	System default language

MORE INFORMATION
================

The following information is taken from the WINNLS.H file.

```
   A language ID is a 16-bit value that is the combination of a
```

primary language ID and a secondary language ID. The bits are allocated as follows:



Language ID creation/extraction macros:

MAKELANGID - Construct language ID from primary language ID and sublanguage ID.
PRIMARYLANGID - Extract primary language ID from a language ID.
SUBLANGID - Extract sublanguage ID from a language ID.

The macros are defined as follows

```
#define MAKELANGID(p, s)       ((((USHORT)(s)) << 10) | (USHORT)(p))
#define PRIMARYLANGID(lgid)    ((USHORT)(lgid) & 0x3ff)
#define SUBLANGID(lgid)        ((USHORT)(lgid) >> 10)
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsRc

FormatMessage() Converts GetLastError() Codes

PSS ID Number: Q94999

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The FormatMessage() application programming interface (API) allows you to convert error codes returned by GetLastError() into error strings, using FORMAT_MESSAGE_FROM_SYSTEM in the dwFlags parameter.

MORE INFORMATION

The following code fragment demonstrates how to get the system message string:

```
LPVOID lpMessageBuffer;

FormatMessage(
  FORMAT_MESSAGE_ALLOCATE_BUFFER |
  FORMAT_MESSAGE_FROM_SYSTEM,
  NULL,
  GetLastError(),
  MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), //The user default language
  (LPTSTR) &lpMessageBuffer,
  0,
  NULL );

//... now display this string

// Free the buffer allocated by the system

LocalFree( lpMessageBuffer );
```

REFERENCES

For more information on language identifiers, please see the topic MAKELANGID in the Win32 Programmer's Reference.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

Fractional Point Sizes Not Supported in ChooseFont()

PSS ID Number: Q77843

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The ChooseFont() common dialog box library routine does not support fractional point sizes. When a fractional point size is entered, it is rounded to the nearest integral point size.

Rounding point sizes affects certain printers that support fractional font sizes. For example, one particular HP LaserJet font cartridge contains an 8.5-point font. The ChooseFont() dialog box displays this font as an 8-point font.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Freeing Memory for Transactions in a DDEML Client App

PSS ID Number: Q83912

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

A Dynamic Data Exchange Management Library (DDEML) client application can request data from a server both synchronously and asynchronously by calling the DdeClientTransaction function.

To make a synchronous request, the client application specifies XTYP_REQUEST as the value for the uType parameter to DdeClientTransaction, and any reasonable value for the uTimeout parameter.

To make an asynchronous request, the client application specifies XTYP_REQUEST as the value for the uType parameter to DdeClientTransaction, and TIMEOUT_ASYNC as the value for the uTimeout parameter.

The client can also establish an advise loop with a server application by specifying XTYP_ADVSTART as the value for the uType parameter. In an advise loop, the client application's callback function receives an XTYP_ADVDATA transaction each time the specified data item changes in the server application. (NOTE: This article discusses only hot advise loops in which changed data is communicated to the application. No data is transferred for a warm advise loop, only a notification that the data changed.)

The client application must free the data handle it receives from a synchronous transaction; however, the client application should not free the data handle it receives from an asynchronous transaction or from an advise loop.

MORE INFORMATION

If the client application initiates a synchronous transaction, the DdeClientTransaction function returns a handle to the requested data. If the client application initiates an asynchronous transaction, the DdeClientTransaction function returns either TRUE or FALSE. When the data becomes available, the DDEML sends the client application an XTYP_XACT_COMPLETE notification accompanied by a handle to the requested data. In an active advise loop, the DDEML sends the client

application an XTYP_ADVDATA notification accompanied by a handle to the updated data.

In the synchronous case, the client application must call DdeFreeDataHandle before it terminates to free a data handle (and the associated memory) that it received as the return value from DdeClientTransaction. If the DDEML server specified HDATA_APPOWNED when it created the data handle, then the data is invalidated when the client calls DdeFreeDataHandle; the server must call DdeFreeDataHandle before terminating to free the associated memory.

In the asynchronous case, the DDEML sends the client application's callback function an XTYP_XACT_COMPLETE notification when the server has completed the transaction. A handle to the requested data accompanies the notification as the hData parameter to the callback function. This handle is valid until control returns from the client application's callback function. Once the client application's callback function returns control, the DDEML may free the data handle and the client application must not assume that the data handle received in the callback function remains valid. This fact has two implications, as follows:

- The client application cannot call DdeFreeDataHandle on the data handle it receives with an XTYP_XACT_COMPLETE transaction. If the client invalidates the data handle by freeing it in the client's callback function, and the DDEML later attempts to free the handle, a Fatal Exit will result.

- The client application must make a local copy of the data it receives with the XTYP_XACT_COMPLETE transaction to use that data after the callback function returns.

In an advise loop, the client application should not free the data handle that it receives as the hData parameter to the callback function. The DDEML frees the data handle when the client application returns from its callback function. If the client calls DdeFreeDataHandle on the data handle, the DDEML will cause a Fatal Exit when it attempts to free the same data handle.

These rules apply to all data handles, whether or not the server application specified the HDATA_APPOWNED flag when it created the handle.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Freeing Memory in a DDEML Server Application

PSS ID Number: Q83413

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

A Dynamic Data Exchange Management Library (DDEML) server application calls the DdeCreateDataHandle function to allocate a block of memory for data it will send to a client application. DdeCreateDataHandle returns a handle to a block of memory that can be passed between applications.

The server application owns every data handle it creates. However, it is not necessary to call DdeFreeDataHandle under every circumstance. This article details the circumstances under which the server application must call DdeFreeDataHandle and when the DDEML will automatically free a data handle.

MORE INFORMATION

If the server application specifies the HDATA_APPOWNED flag in the afCmd parameter to DdeCreateDataHandle, it must explicitly call DdeFreeDataHandle to free the memory handle. Using HDATA_APPOWNED data handles is convenient when data, such as system topic information, is likely to be passed to a client application more than once, because the server calls DdeCreateDataHandle only once, regardless of the number of times the data handle is passed to a client application.

When it closes down, a server application must call DdeFreeDataHandle for each data handle that it has not passed to a client application. When the server creates a handle without specifying HDATA_APPOWNED, and passes the handle to a client application in an asynchronous transaction, the DDEML frees the data handle when the client returns from its callback function. Therefore, the server is not required to free the data handle it passes to a client because the DDEML frees the handle. However, if the data handle is never sent to a client application, the server must call DdeFreeDataHandle to free the handle. It is the client application's responsibility to call DdeFreeDataHandle for any data provided by a server in a synchronous transaction.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui

KBSubcategory: UsrDde

Freeing PackDDElParam() Memory

PSS ID Number: Q94149

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

When posting DDE messages via PostMessage(), an application first calls PackDDElParam() and sends its return value (a pointer cast to LPARAM) as the lParam in PostMessage().

Normally the receiving application is responsible for freeing the structure [via FreeDDElParam()]. However, if the call to PostMessage() fails, the posting application must free the packed data. This is also the method used by 16-bit Windows.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Gaining Access to ACLs

PSS ID Number: Q102098

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

To gain access to a security access control list (SACL), a process must have the SE_SECURITY_NAME privilege. When requesting access, the calling process must request ACCESS_SYSTEM_SECURITY in the desired access mask.

There is not a privilege that controls read or write access to a discretionary access control list (DACL). Instead, access to read and write an object's DACL is granted by the READ_CONTROL and WRITE_DAC access rights, respectively. These rights must be specifically granted to the user (or group containing the user) for DACL read or write access to be granted. If the owner of an object requests READ_CONTROL or WRITE_DAC, the access will always be granted.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

GDI Objects and Windows 95 Heaps

PSS ID Number: Q125699

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

Under Windows version 3.1, GDI allocates all resources from a single 64K heap. This limit has caused many applications to run out of GDI resources, especially when using objects that can really take up a lot of the heap, like elliptical regions. This caused GDI resources to be dangerously low when executing several applications at once.

Windows 95 has now introduced a combination of a 16-bit heap and an additional 32-bit heap. The 16-bit heap is still limited to 64K but the 32-bit heap can grow as large as available memory.

As in Windows version 3.1, the 16-bit GDI.EXE of Windows 95 continues to have a 16-bit DGROUP segment with a local heap within it, and most logical objects are still stored in this local heap. The data structures that describe brushes, bitmap headers, and pens, for example, stay in the 16-bit heap. All physical objects, like fonts and bitmaps, are now stored in the 32-bit heap. GDI regions have also been moved to the 32-bit heap. Moving these GDI resources to the 32-bit heap takes the pressure off of the 64K 16-bit heap.

Regions can take up a large amount of resources and were the main source of problems with GDI memory in Windows version 3.1. This will not be a limitation in Windows 95 because regions are stored in the 32-bit heap. Applications will be able to use much more complex regions, and regions will be more useful now that they are not limited to a local 64K heap.

Windows 95, like Windows NT, will free all GDI resources owned by a 32-bit process when that process terminates. Windows 95 will also clean up any GDI resources of 16-bit processes marked as a 4.0 application. Because GDI objects were sharable between applications in Windows version 3.1, Windows 95 will not immediately clean up GDI resources for 16-bit applications marked with a version less than 4.0. However, when all 16-bit applications have finished running, all GDI resources allocated by previous 16-bit applications will be cleaned up.

Additional reference words: 4.00 Heaps
KBCategory: kbgraphic
KBSubcategory: GdiGeneral

General Overview of Win32s

PSS ID Number: Q83520

The information in this article applies to:

- Microsoft Win32s versions 1.1 and 1.2

SUMMARY
=======

The following is intended as a general introduction to Win32s. More information can be found in the "Win32s Programmer's Reference" and by querying for Knowledge Base articles on "Win32s".

MORE INFORMATION
================

General Overview

The Win32 API consists of the Window 3.1 (Win16) API, with types stretched to 32 bits, plus the addition of APIs which offer new functionality, like threads, security, services, and virtual memory. Applications developed using the Win32 API are called "Win32-based applications".

Win32s is a set of DLLs and a VxD which allow Win32-based applications to run on top of Windows or Windows for Workgroups version 3.1. Win32s supports a subset of the Win32 API, some directly (like memory management) and some through thunks to the 16-bit systems (particularly GDI and User). Win32s contains function stubs for the APIs that are not supported, which return ERROR_NOT_IMPLEMENTED. Win32s also includes 4 new APIs which support the Universal Thunk (UT). For details on which API are supported under Win32s, refer to the individual API entries in the help file "Win32 API Reference." Among the new features gained from Win32 are structured exception handling (SEH), FP emulation, memory-mapped files, named shared memory, and sparse memory.

Win32-based applications running on Windows 3.1 will generally be faster than their Win16 equivalents on Windows 3.1, particularly if they are memory or floating-point intensive. The actual speed improvement varies with each application, because it depends on how often you cross the thunk layer. Each call which uses a thunk is no more than 10 percent slower than a direct call.

Binary Compatibility

Win32s offers binary compatibility for Win32-based applications on Windows 3.1 and Windows NT.

When you call a Win32 API, two options should be allowed:

- Option A: Your code should allow for a successful return from the

function call.

- Option B: Your code should allow for an unsuccessful return from the function call.

For example, if the application is running under Windows 3.1 and a call is made to one of the supported APIs, then the call returns successfully and option A is executed. If the call is made while running under Windows NT, the call again returns successfully and option A should be executed. However, if running under Windows 3.1 and a Win32 API function is called that is unsupported, then an error code is returned and option B should be executed.

If, for example, option A were using a CreateThread() call, then option B would be alternative code, which would handle the task using a single-thread solution.

Programming Issues

Win32-based applications cannot use MS-DOS and BIOS interrupts; therefore, the Win32s VxD has Win32 entries for each Interrupt 21 and the BIOS calls.

The Win32s DLLs may thunk to Win16 when a Win32 application makes a call. The 32-bit parameters are copied from the 32-bit stack to a 16-bit stack and the 16-bit entry point is called. The Win32 application has a 128K stack. When switching to the 16-bit side via UT, the same stack is used, and a 16:16 stack pointer is created which points to the top of the stack. The selector base is set so that there is at least an 8K stack for the 16-bit code.

There are other semantic difference between Windows 3.1 and Win32. Windows 3.1 will run applications for Win32 nonpreemptively in a single, shared address space, while Windows NT runs them preemptively in separate address spaces. It is therefore important that you test your Win32-based application on both Windows 3.1 and Windows NT.

If you need to call routines that reside in a 16-bit DLL or Windows from 32-bit code, you can do this using the Win32s Universal Thunk or other client-server techniques. For a description of UT, please see the "Win32s Programmer's Reference" and the sample UTSAMPLE.

DDE, OLE, WM_COPYDATA, the clipboard, metafiles, and bitmaps can be used between 16-bit Windows-based and Win32-based applications on both Windows 3.1 and Windows NT. RPC is not supported from Win32-based applications.

Additional reference words: 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

GetClientRect() Coordinates Are Not Inclusive

PSS ID Number: Q43596

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The coordinates returned by GetClientRect() are not inclusive. For example, to draw a border around the edge of the client area, draw it from the coordinates (Rectangle.left, Rectangle.top) to (Rectangle.right-1, Rectangle.bottom-1).

Addtional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrPnt

GetCommandLine() Under Win32s

PSS ID Number: Q102762

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

Under Win32s, GetCommandLine() includes the full drive/path of the executable, while under Windows NT GetCommandLine() does not include the full path.

When programs are run from the Program Manager or the File Manager on Windows 3.1, they are spawned using the full path. As a result, argv[0] will have the complete path. When a Win32s application is spawned by a 16-bit application, Windows detects that the application is a Win32s application. The full path is passed to Win32s regardless of whether or not WinExec() was invoked with the full path. As a result, 32-bit applications receive the full path.

When a Win32-based application is spawned from another Win32-based application, the 32-bit kernel passes the information as given by the parent process [that is, if a Win32-based application is started via CreateProcess() from another Win32-based application, argv[0] will contain the path that the spawning application passed in].

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

GetCurrentTime and GetTickCount Functions Identical

PSS ID Number: Q45702

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The GetCurrentTime() and GetTickCount() functions are identical. Each returns the number of milliseconds (+/- 55 milliseconds) since the user started Windows, and the return value of each function is declared to be a DWORD.

The following code demonstrates these two functions:

```
   DWORD dwCurrTime;
   DWORD dwTickCount;
   char szCurrTime[50];

   dwCurrTime = GetCurrentTime ();
   dwTickCount = GetTickCount ();

   sprintf (szCurrTime, "Current time = %lu\nTick count = %lu",
            dwCurrTime, dwTickCount);

   MessageBox (hWnd, szCurrTime, "Times", MB_OK);
```

NOTE: GetCurrentTime() and GetTickCount() return an unsigned double word (32-bit DWORD) which gives a maximum count of 4,294,967,296 ticks. This number will yield a maximum count of 49.71 days which the system keeps track of running time.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrTim

GetDeviceCaps(hDC, RASTERCAPS) Description

PSS ID Number: Q75912

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

GetDeviceCaps(hDC, RASTERCAPS) returns the raster capabilities bit field in the GDIINFO structure, which indicates the raster capabilities of the device. The RASTERCAPS index of the GetDeviceCaps() function is documented in the "Microsoft Windows Software Development Kit Reference Volume 1" on page 4-168. The flags available include: RC_BANDING, RC_BITBLT, RC_BITMAP64, RC_DI_BITMAP, RC_DIBTODEV, RC_GDI20_OUTPUT, RC_PALETTE, RC_SCALING, RC_STRETCHBLT, and RC_STRETCHDIB. The GDIINFO structure itself is documented in the "Microsoft Windows Device Development Kit Device Driver Adaptation Guide."

An application should use GetDeviceCaps() to query the printer driver for device capabilities. For example, before printing a bitmap larger than 64K, the application should query the driver using GetDeviceCaps() with the index RASTERCAPS and the flag RC_BITMAP64. If the application fails to test for the capability and prints a bitmap larger than 64K, unexpected printer output may occur if the driver does not support bitmaps larger than 64K.

In particular, if the driver does not support a capability, GDI will attempt to simulate it using a more fundamental capability of the driver. However, the resulting GDI simulation is usually slower, is of lower quality, or differs in some way from a device driver implementation of the capability.

Field Name	Capability or Function	GDI Call to Invoke Capability or Function	Differences in Functionality Resulting from Supporting or not Supporting Capability
0	BitBlt		?
1	Requires banding		?
2	Requires scaling		?

3	Supports >64K bitmaps		?
4	Supports ExtTextOut, FastBorder, GetCharWidth	ExtTextOut	GDI will call StrBlt() once for each character to simulate the ExtTextOut() function's ability to position proportionally-spaced characters. This can be very slow. GDI simulates bold text by overstriking one or more times. This fails on laser printers. Laser printer drivers that support ExtTextOut() offset the text before overstriking.
		FastBorder	
		GetCharWidth	Returns 0 if driver does not support GetCharWidth; otherwise, it calls ExtTextOut() or StrBlt() with count = -1 to obtain the width of each individual character.
5	Has state block		
6	Saves bitmaps in shadow memory		
7	RC_DI_BITMAP	Supports Get and Set DIBs and RLEs	If GDI is called upon to copy a RLE bitmap that contains a transparent window (region not defined by the bitmap), the window will be filled by the current background color. The destination for any GDI DIB operation is a monochrome bitmap. Unidrv offers a variety of halftone dithering techniques to simulate a range of intensities on black and white and color printers. GDI does not. Unidrv offers intensity

			adjustment to darken and lighten halftone output. Unidrv does not support RLE DIBs at present.
8	RC_PALETTE	Performs color palette management	
9	RC_DIBTODEV	Supports SetDIBitsToDevice	See comments above for RC_DI_BITMAP
10	RC_BIGFONTS	Supports Windows 3.0 FONTINFO structure format	This flag specifies the format of the FONTINFO structure passed between GDI's SelectObject() call and the driver's RealizeFont() function. If a bit is set, the Windows 3.0 format is used. Otherwise, the Windows 2.0 font file format is used.
11	RC_STRETCHBLT	Supports StretchBlt	
12	RC_FLOODFILL	Supports flood fill	
13	RC_STRETCHDIB	Supports StretchDIBits	See comments above for RC_DI_BITMAP. Additionally, Unidrv has these limits on the degree of stretching and shrinking supported: The X and Y axes may be scaled independently. The maximum scale-up factor is 256. The maximum product of the X and Y scale-down factors is 256. Therefore, if both axes are scaled down equally, the maximum scale-down factor for each axis is 16.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiDc

GetGlyphOutline() Native Buffer Format

PSS ID Number: Q87115

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The GetGlyphOutline function provides a method for an application to retrieve the lowest-level information about a glyph in the TrueType environment. This article describes the format of the data the GetGlyphOutline function returns.

MORE INFORMATION

A glyph outline is a series of contours that describe the glyph. Each contour is defined by a TTPOLYGONHEADER data structure, which is followed by as many TTPOLYCURVE data structures as are required to describe the contour.

Each position is described by a POINTFX data structure, which represents an absolute position, not a relative position. The starting and ending point for the glyph is given by the pfxStart member of the TTPOLYGONHEADER data structure.

The TTPOLYCURVE data structures fall into two types: a TT_PRIM_LINE record or a TT_PRIM_QSPLINE record. A TT_PRIM_LINE record is a series of points; lines drawn between the points describe the outline of the glyph. A TT_PRIM_QSPLINE record is a series of points defining the quadratic splines (q-splines) required to describe the outline of the character.

In TrueType, a q-spline is defined by three points (A, B, and C), where points A and C are on the curve and point B is off the curve. The equation for each q-spline is as follows (xA represents the x-coordinate of point A, yA represents the y-coordinate of point A, and so on)

```
x(t) = (xA-2xB+xC)*t^2 + (2xB-2xA)*t + xA
y(t) = (yA-2yB+yC)*t^2 + (2yB-2yA)*t + yA
```

where t varies from 0.0 to 1.0.

The format of a TT_PRIM_QSPLINE record is as follows:

- Point A on the q-spline is the current position (either pfxStart in the TTPOLYGONHEADER, the starting point for the TTPOLYCURVE, or the ending point of the previous TTPOLYCURVE).

- Point B is the current point in the record.

- Point C is as follows:

    - If the record has two or more points following point B, point C is the midpoint between point B and the next point in the record.
    - Otherwise, point C is the point following point B.

The following code presents the algorithm used to process a TT_PRIM_QSPLINE record. While this code demonstrates how to extract q-splines from a TT_PRIM_QSPLINE record, it is not appropriate for use in an application.

```
pfxA = pfxStart;                // Starting point for this polygon

for (u = 0; u < cpfx - 1; u++)  // Walk through points in spline
   {
   pfxB = apfx[u];              // B is always the current point
   if (u < cpfx - 2)            // If not on last spline, compute C
      {
      pfxC.x = (pfxB.x + apfx[u+1].x) / 2;  // x midpoint
      pfxC.y = (pfxB.y + apfx[u+1].y) / 2;  // y midpoint
      }
   else                         // Else, next point is C
      pfxC = apfx[u+1];

                                // Draw q-spline
   DrawQSpline(hdc, pfxA, pfxB, pfxC);
   pfxA = pfxC;                 // Update current point
   }
```

The algorithm above manipulates points directly, using floating-point operators. However, points in q-spline records are stored in a FIXED data type. The following code demonstrates how to manipulate FIXED data items:

```
FIXED fx;
long *pl = (long *)&fx;

// Perform all arithmetic on *pl rather than on fx
*pl = *pl / 2;
```

The following function converts a floating-point number into the FIXED representation:

```
FIXED FixedFromDouble(double d)
{
   long l;
```

```
   l = (long) (d * 65536L);
   return *(FIXED *)&l;
}
```

In a production application, rather than writing a DrawQSpline function to draw each q-spline individually, it is more efficient to calculate points on the q-spline and store them in an array of POINT data structures. When the calculations for a glyph are complete, pass the POINT array to the PolyPolygon function to draw and fill the glyph.

The following example presents the data returned by the GetGlyphOutline for the lowercase "j" glyph in the 24-point Arial font of the 8514/a (Small Fonts) video driver:

```
GetGlyphOutline GGO_NATIVE 'j'
  dwrc            = 208      // Total native buffer size in bytes
  gmBlackBoxX, Y  = 6, 29    // Dimensions of black part of glyph
  gmptGlyphOrigin = -1, 23   // Lower-left corner of glyph
  gmCellIncX, Y   = 7, 0     // Vector to next glyph origin

TTPOLYGONHEADER #1           // Contour for dot on "j"
  cb       = 44              // Total size of dot polygon
  dwType   = 24              // TT_POLYGON_TYPE
  pfxStart = 2.000, 20.000   // Start at lower-left corner of dot

  TTPOLYCURVE #1
    wType  = TT_PRIM_LINE
    cpfx   = 3
    pfx[0] = 2.000, 23.000
    pfx[1] = 5.000, 23.000
    pfx[2] = 5.000, 20.000   // Automatically close to pfxStart


TTPOLYGONHEADER #2   // Contour for body of "j"
  cb       = 164     // Total size is 164 bytes
  dwType   = 24      // TT_POLYGON_TYPE
  pfxStart = -1.469, -5.641

  TTPOLYCURVE #1     // Finish flat bottom end of "j"
    wType  = TT_PRIM_LINE
    cpfx   = 1
    pfx[0] = -0.828, -2.813

  TTPOLYCURVE #2    // Make hook in "j" with spline
                    // Point A in spline is end of TTPOLYCURVE #1
    wType  = TT_PRIM_QSPLINE
    cpfx   = 2              // two points in spline -> one curve
    pfx[0] = -0.047, -3.000  // This is point B in spline
    pfx[1] = 0.406, -3.000   // Last point is always point C

  TTPOLYCURVE #3    // Finish hook in "j"
                    // Point A in spline is end of TTPOLYCURVE #2
    wType  = TT_PRIM_QSPLINE
    cpfx   = 3              // Three points -> two splines
```

```
    pfx[0] = 1.219, -3.000   // Point B for first spline
                             // Point C is (pfx[0] + pfx[1]) / 2
    pfx[1] = 2.000, -1.906   // Point B for second spline
    pfx[2] = 2.000, 0.281    // Point C for second spline

  TTPOLYCURVE #4    // Majority of "j" outlined by this polyline
    wType  = TT_PRIM_LINE
    cpfx   = 3
    pfx[0] = 2.000, 17.000
    pfx[1] = 5.000, 17.000
    pfx[2] = 5.000, -0.250

  TTPOLYCURVE #5    // start of bottom of hook
    wType  = TT_PRIM_QSPLINE
    cpfx   = 2               // One spline in this polycurve
    pfx[0] = 5.000, -3.266   // Point B for spline
    pfx[1] = 4.188, -4.453   // Point C for spline

  TTPOLYCURVE #6    // Middle of bottom of hook
    wType  = TT_PRIM_QSPLINE
    cpfx   = 2               // One spline in this polycurve
    pfx[0] = 3.156, -6.000   // B for spline
    pfx[1] = 0.766, -6.000   // C for spline

  TTPOLYCURVE #7    // Finish bottom of hook and glyph
    wType  = TT_PRIM_QSPLINE
    cpfx   = 2               // One spline in this polycurve
    pfx[0] = -0.391, -6.000  // B for spline
    pfx[1] = -1.469, -5.641  // C for spline

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiTt
```

GetInputState Is Faster Than GetMessage or PeekMessage

PSS ID Number: Q35605

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

This article describes a method to quickly determine whether an application for the Microsoft Windows graphical environment has any keyboard or mouse messages in its queue without calling the GetMessage or PeekMessage functions.

NOTE: In Win32, GetInputState is thread-local only.

MORE INFORMATION
================

The GetInputState function returns this information more quickly than GetMessage or PeekMessage. GetInputState returns TRUE (nonzero) if either a keyboard or mouse message is in the application's input queue. If the application must distinguish between a mouse and a keyboard message, GetInputState returns the value 2 for a keyboard and the value 1024 for a mouse message.

Because difficulties may arise if the application loses the input focus, use GetInputState only in tight loop conditions where execution speed is critical.

In Win32, message queues are not global as they are in 16-bit Windows. The message queues are local to the thread. When you call GetInputState, you are checking to see if there are mouse or keyboard messages for the calling thread only. If a window created by another thread in the application has the keyboard input waiting, GetInputState will not be able to check for those messages.

Additional reference words: 3.00 3.10 3.50 4.00 95 yield
KBCategory: kbui
KBSubcategory: UsrMsg

GetLastError() May Differ Between Windows 95 and Windows NT

PSS ID Number: Q127991

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

The extended error codes returned by the GetLastError() API are not guaranteed to be the same under Windows 95 and Windows NT. This difference applies to extended error codes generated by calls to GDI, Window Management, and System Services APIs.

The set of potential error codes returned by any particular Win32 API depends on many underlying components, including system kernel-mode components and loaded drivers. The extended error codes are not a part of the Win32 specification. Therefore, they can change as operating system and driver code is changed.

It is impossible to get the error codes for each API in synch across operating system and platforms. Windows NT and Windows 95 have different code bases. Third-party drivers return error codes that are mapped to Win32 error codes. In addition, it would be difficult to accurately maintain error code information in the Win32 Programmer's Reference. Therefore, this information is not included in the documentation.

In general, you should not rely on GetLastError() returning the same values under Windows 95 and Windows NT. Sometimes, an API will fail and GetLastError() will return 0 (ERROR_SUCCESS) under Windows 95. This is because some of the APIs do not set error codes under Windows 95.

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: BseMisc UsrMisc GdiMisc

GetSystemMetrics(SM_CMOUSEBUTTONS) Fails Under Win32s

PSS ID Number: Q124836

The information in this article applies to:

- Microsoft Win32s versions 1.15, 1.15a, and 1.20

Under Win32s, the Win32 API GetSystemMetrics() is implemented as a direct thunk to Windows version 3.1. Therefore, the call will return whatever Windows version 3.1 would return for a similar call to the Windows API GetSystemMetrics().

GetSystemMetrics() returns zero for all unrecognized parameters. Under Windows, this includes the new flag SM_CMOUSEBUTTONS, which was introduced in the Win32 API. Therefore, avoid using the SM_CMOUSEBUTTONS flag when your Win32-based application is running under Win32s.

Additional reference words: 1.20
KBCategory: kbprg
KBSubcategory: W32s

Getting a Dialog to Use an Icon When Minimized

PSS ID Number: Q114612

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

The standard Windows dialog box does not have an icon when it is minimized. A dialog box can be made to use an icon by replacing the standard dialog box class with a private dialog box class.

MORE INFORMATION

================

The standard dialog box class specifies NULL as the value of the hIcon field of its WNDCLASS structure. So no icon is drawn when the standard dialog box is minimized.

An icon can be specified by getting the dialog to use a private class as follows:

1. Register a private class.

```
WNDCLASS wc;

wc.style = CS_DBLCLKS | CS_SAVEBITS | CS_BYTEALIGNWINDOW;
wc.lpfnWndProc = DefDlgProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = DLGWINDOWEXTRA;
wc.hInstance = hinst;
wc.hIcon = LoadIcon(hinst, "DialogIcon");
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = COLOR_WINDOW + 1;
wc.lpszMenuName = NULL;
wc.lpszClassName = "MyDlgClass";
RegisterClass(&wc);
```

   NOTE: The default dialog window procedure, DefDlgProc(), is used as the window procedure of the class. This causes windows of this class to behave as standard dialogs. The cbWndExtra field has to be assigned to DLGWINDOWEXTRA - the dialog box stores its internal state information in these extra window bytes. The icon to be used when the dialog box is minimized is assigned to the hIncon field.

2. Get the dialog box to use the private class.

Use the CLASS statement in the dialog box template to get the dialog box to use the private class:

```
IDD_MYDIALOG DIALOG 0, 0, 186, 92
CLASS "MyDlgClass"
:
```

3. Create the dialog box using DialogBox() or CreateDialog().

```
DialogBox (hinst,
           MAKEINTRESOURCE (IDD_MYDIALOG),
           NULL,
          (DLGPROC)MyDlgFunc);
```

MyDlgFunc() is the dialog function implemented by the application. When the dialog box is minimized, it will use the icon specified in the private class.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Getting and Using a Handle to a Directory

PSS ID Number: Q105306

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

CreateDirectory() can be used to open a new directory. An existing directory can be opened by calling CreateFile(). To open an existing directory with CreateFile(), it is necessary to specify the flag FILE_FLAG_BACKUP_SEMANTICS. The following code shows how this can be done:

```
   HANDLE hFile;

   hFile = CreateFile( "c:\\mstools",
               GENERIC_READ | GENERIC_WRITE,
               FILE_SHARE_READ | FILE_SHARE_WRITE,
               NULL,
               OPEN_EXISTING,
               FILE_FLAG_BACKUP_SEMANTICS,
               NULL
           );
   if( hFile == INVALID_HANDLE_VALUE )
      MessageBox( NULL, "CreateFile() failed", NULL, MB_OK );
```

The handle obtained can be used to obtain information about the directory or to set information about the directory. For example:

```
   BY_HANDLE_FILE_INFORMATION fiBuf;
   FILETIME ftBuf;
   SYSTEMTIME stBuf;
   char msg[40];

   GetFileInformationByHandle( hFile, &fiBuf );
   FileTimeToLocalFileTime( &fiBuf.ftLastWriteTime, &ftBuf );
   FileTimeToSystemTime( &ftBuf, &stBuf );
   wsprintf( msg, "Last write time is %d:%d  %d/%d/%d",
      stBuf.wHour,stBuf.wMinute,stBuf.wMonth,stBuf.wDay,stBuf.wYear );
   MessageBox( NULL, msg, NULL, MB_OK );
```

MORE INFORMATION

================

Opening directories with CreateFile is not supported on Windows 95.

This code does not work on Win32s, because MS-DOS does not support opening a directory. If you are looking for the creation time of a directory, use FindFirstFile(), because it works on all platforms.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

Getting Floppy Drive Information

PSS ID Number: Q115828

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

To get the media type(s) supported by a floppy drive, it is necessary to call CreateFile() to get a handle to the drive and then DeviceIoControl() to get the information. However, if there is no floppy disk in the floppy drive, the following message box may appear when CreateFile() is called for drive A (\\.\a:):

   There is no disk in the drive. Please insert a disk into drive A:

When calling CreateFile(), be sure to use 0 for the access mode and FILE_SHARE_READ for the share mode so that the user will not be prompted to insert a floppy disk:

```
   CreateFile(
      szFileName,
      0,
      FILE_SHARE_READ,
      NULL,
      OPEN_ALWAYS,
      0,
      NULL
   );
```

Another way to avoid the message box prompt is to put

```
   SetErrorMode( SEM_FAILCRITICALERRORS );
```

before the call to CreateFile().

MORE INFORMATION
================

The following sample code is based on the code in the FLOPPY SDK sample, but it simply displays the media type(s) supported by floppy drive A. The code demonstrates one way to retrieve the supported media without requiring you to insert a floppy disk in the drive.

Sample Code

```
#include <windows.h>
#include <stdio.h>
```

```
#include <winioctl.h>

DISK_GEOMETRY SupportedGeometry[20];
DWORD SupportedGeometryCount;

VOID
GetSupportedGeometrys( HANDLE hDisk )
{
   DWORD ReturnedByteCount;

   if( DeviceIoControl(
          hDisk,
          IOCTL_DISK_GET_MEDIA_TYPES,
          NULL,
          0,
          SupportedGeometry,
          sizeof(SupportedGeometry),
          &ReturnedByteCount,
          NULL
      ))
      SupportedGeometryCount = ReturnedByteCount /  sizeof(DISK_GEOMETRY);

   else SupportedGeometryCount = 0;
}

VOID
PrintGeometry( PDISK_GEOMETRY lpGeometry )
{
   LPSTR MediaType;

   switch ( lpGeometry->MediaType ) {
      case F5_1Pt2_512:
         MediaType = "5.25, 1.2MB,  512 bytes/sector";
         break;
      case F3_1Pt44_512:
         MediaType = "3.5,  1.44MB, 512 bytes/sector";
         break;
      case F3_2Pt88_512:
         MediaType = "3.5,  2.88MB, 512 bytes/sector";
         break;
      case F3_20Pt8_512:
         MediaType = "3.5,  20.8MB, 512   bytes/sector";
         break;
      case F3_720_512:
         MediaType = "3.5,  720KB,  512 bytes/sector";
         break;
      case F5_360_512:
         MediaType = "5.25, 360KB,  512 bytes/sector";
         break;
      case F5_320_512:
         MediaType = "5.25, 320KB,  512 bytes/sector";
         break;
      case F5_320_1024:
         MediaType = "5.25, 320KB,  1024 bytes/sector";
         break;
```

```
      case F5_180_512:
         MediaType = "5.25, 180KB,  512 bytes/sector";
         break;
      case F5_160_512:
         MediaType = "5.25, 160KB,  512  bytes/sector";
         break;
      case RemovableMedia:
         MediaType = "Removable media other than floppy";
         break;
      case FixedMedia:
         MediaType = "Fixed hard disk media";
         break;
      default:
         MediaType = "Unknown";
         break;
   }
   printf("    Media Type %s\n", MediaType );
   printf("    Cylinders %d, Tracks/Cylinder %d, Sectors/Track %d\n",
          lpGeometry->Cylinders.LowPart, lpGeometry->TracksPerCylinder,
          lpGeometry->SectorsPerTrack
         );
}

void main( int argc, char *argv[], char *envp[] )
{
   HANDLE hDrive;
   UINT i;

   hDrive = CreateFile(
                  "\\\\.\\a:",
                  0,
                  FILE_SHARE_READ,
                  NULL,
                  OPEN_ALWAYS,
                  0,
                  NULL
                  );
   if ( hDrive == INVALID_HANDLE_VALUE )
   {
      printf( "Open failed: %d\n", GetLastError() );
      ExitProcess(1);
   }

   GetSupportedGeometrys( hDrive );

   printf( "\nDrive A supports the following disk geometries\n" );
   for( i=0; i<SupportedGeometryCount; i++ )
   {
       printf("\n");
       PrintGeometry( &SupportedGeometry[i] );
   }
   printf("\n");
}

Additional reference words: 3.10 3.50
```

KBCategory: kbprg
KBSubcategory: BseMisc

Getting Real Handle to Thread/Process Requires Two Calls

PSS ID Number: Q90470

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

The API GetCurrentThread() returns a pseudo-handle rather than the real handle to the thread. To get the real handle to the thread, you need to use DuplicateHandle() using the pseudo-handle that is returned from GetCurrentThread(). In addition, to get the real handle to a process, you need to call DuplicateHandle() after calling GetCurrentProcess().

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseProcThrd

Getting Resources from 16-Bit DLLs Under Win32s

PSS ID Number: Q105761

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

A Win32-based application running under Win32s can load a 16-bit dynamic-link library (DLL) using LoadLibrary() and free it with FreeLibrary(). This behavior is allowed primarily so that GetProcAddress() can be called for printer driver application programming interfaces (APIs).

Calling FindResource() with the handle that LoadLibrary() returns to the DLL that it just loaded results in an access violation. However, the Win32-based application can use the following APIs with this handle

   LoadBitmap
   LoadCursor
   LoadIcon

because this results in USER.EXE (16-bit) making calls to KERNEL.EXE.

If you go through a Universal Thunk to get raw resource data from the 16-bit DLL, it is necessary to convert the resource to 32-bit format, because the resource format is different from the 16-bit format. The 32-bit format is described in the Software Development Kit (SDK) file DOC\SDK\FILEFRMT\RESFMT.TXT.

To determine whether a DLL is a 32-bit or 16-bit DLL, check the DLL header. The DWORD at offset 0x3C indicates where to look for the PE signature. Compare the 4 bytes there to 0x00004550 to determine whether this is a Win32 DLL.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Getting the Filename Given a Window Handle

PSS ID Number: Q119163

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

To find the filename of the program that created a given window under Windows, you would use GetWindowLong(hWnd, GWL_HINSTANCE) to find the module handle and then GetModuleFileName() to find the filename. This method cannot be used under Windows NT because instance handles are not global, but are unique to the address space in which the application is running.

If the application that created the window is a Windows-based application, the name returned is "ntvdm". To get the actual filename, you need to spawn a Win16 application that will call GetModuleFileName() and pass the filename back to your application using some form of interprocess communication (IPC).

MORE INFORMATION

================

To find the filename of an application once you have its window handle, first use GetWindowThreadProcessId() to find the process ID (PID) of the process that created the window. Using the PID, query the registry for the performance data associated with the process. To do this, you have to enumerate all processes in the system, comparing each PID to the PID of the process that you are looking for, until the data for that process is found. (This data includes the name of the process.)

The following sample code demonstrates how to find the filename of the Program Manager, PROGMAN.EXE, after obtaining its window handle:

Sample Code

```
#include <windows.h>
#include <stdio.h>
#include <string.h>

#define Key "SOFTWARE\\Microsoft\\Windows
NT\\CurrentVersion\\Perflib\\009"

void GetIndex( char *, char * );
void DisplayFilename( DWORD );

/*****************************************************************\
```

```
* Function: void main( )                                                *
*                                                                       *
* Purpose : Application entry point                                     *
*                                                                       *
\***********************************************************************/

void main( )
{
   HWND                     hWnd;
   DWORD                    dwActiveProcessId;

   // Get window handle of Program Manager's main window.

   hWnd = FindWindow( "Progman", NULL );

   // Get PID of Program Manager.

   GetWindowThreadProcessId( hWnd, &dwActiveProcessId );

   // Display name of Program Manager's executable file.

   printf( "Searching for filename of Program Manager...\n" );
   DisplayFilename( dwActiveProcessId );
}

/***********************************************************************\
* Function: void DisplayFilename( DWORD )                               *
*                                                                       *
* Purpose : Display executable filename of the process whose PID        *
*           is passed in as a parameter.                                *
*                                                                       *
* Comment : The information is retrieved from the performance           *
*           data in the registry.                                       *
*                                                                       *
\***********************************************************************/

void DisplayFilename( DWORD dwProcessId )
{
   DWORD                    CurrentProcessId;
   BOOL                     bContinue = TRUE;
   char                     szIndex[256] = "";
   DWORD                    dwBytes = 12000;
   DWORD                    dwProcessIdOffset;
   int                      i;

   PPERF_DATA_BLOCK         pdb;
   PPERF_OBJECT_TYPE        pot;
   PPERF_INSTANCE_DEFINITION pid;
   PPERF_COUNTER_BLOCK      pcb;
   PPERF_COUNTER_DEFINITION  pcd;

   // Get the index for the PROCESS object.
   GetIndex( "Process", szIndex );

   // Get memory for PPERF_DATA_BLOCK.
```

```
   pdb = (PPERF_DATA_BLOCK) HeapAlloc( GetProcessHeap(),
                                       HEAP_ZERO_MEMORY,
                                       dwBytes);

   // Get performance data.
   while( RegQueryValueEx(HKEY_PERFORMANCE_DATA, (LPTSTR)szIndex, NULL,
                          NULL, (LPBYTE)pdb, &dwBytes) ==
ERROR_MORE_DATA )
   {
      // Increase memory.
      dwBytes += 1000;

      // Allocated memory is too small; reallocate new memory.
      pdb = (PPERF_DATA_BLOCK) HeapReAlloc( GetProcessHeap(),
                                            HEAP_ZERO_MEMORY,
                                            (LPVOID)pdb,
                                            dwBytes);
   }

   // Get PERF_OBJECT_TYPE.
   pot = (PPERF_OBJECT_TYPE)((PBYTE)pdb + pdb->HeaderLength);

   // Get the first counter definition.
   pcd = (PPERF_COUNTER_DEFINITION)((PBYTE)pot + pot->HeaderLength);

   // Get index value for ID_PROCESS.
   szIndex[0] = '\0';
   GetIndex( "ID Process", szIndex );

   for( i=0; i< (int)pot->NumCounters; i++ )
   {
      if (pcd->CounterNameTitleIndex == (DWORD)atoi(szIndex))
      {
         dwProcessIdOffset = pcd->CounterOffset;
         break;
      }

      pcd = ((PPERF_COUNTER_DEFINITION)((PBYTE)pcd + pcd->ByteLength));
   }

   // Get the first instance of the object.
   pid = (PPERF_INSTANCE_DEFINITION)((PBYTE)pot + pot-
>DefinitionLength);

   // Get the name of the first process.
   pcb = (PPERF_COUNTER_BLOCK) ((PBYTE)pid + pid->ByteLength );
   CurrentProcessId = *((DWORD *) ((PBYTE)pcb + dwProcessIdOffset));

   // Find the process object for PID passed in, then print its
   // filename.

   for( i = 1; i < pot->NumInstances && bContinue; i++ )
   {
      if( CurrentProcessId == dwProcessId )
      {
```

```
         printf( "The filename is %ls.exe.\n",
                 (char *) ((PBYTE)pid + pid->NameOffset) );
         bContinue = FALSE;
      }
      else
      {
         pid = (PPERF_INSTANCE_DEFINITION) ((PBYTE)pcb + pcb-
>ByteLength);
         pcb = (PPERF_COUNTER_BLOCK) ((PBYTE)pid + pid->ByteLength);
         CurrentProcessId = *((DWORD *)((PBYTE)pcb +
dwProcessIdOffset));
      }
   }
   if( bContinue == TRUE )
      printf( "Not found.\b" );

   // Free the allocated memory.
   if( !HeapFree(GetProcessHeap(), 0, (LPVOID)pdb) )
      printf( "HeapFree failed in main.\n" );

   // Close handle to the key.
   RegCloseKey( HKEY_PERFORMANCE_DATA );
}

/*****************************************************************\
* Function: void GetIndex( char *, char * )                      *
*                                                                *
* Purpose : Get the index for the given counter                  *
*                                                                *
* Comment : The index is returned in the parameter szIndex       *
*                                                                *
\*****************************************************************/

void GetIndex( char *pszCounter, char *szIndex )
{
   char*  pszBuffer;
   char*  pszTemp;
   char   szObject[256] = "";
   DWORD  dwBytes;
   HANDLE hKeyIndex;
   int    i = 0;
   int    j = 0;

   // Open the key.
   RegOpenKeyEx( HKEY_LOCAL_MACHINE,
                 Key,
                 0, KEY_READ,
                 &hKeyIndex );

   // Get the size of the counter.
   RegQueryValueEx( hKeyIndex,
                    "Counters",
                    NULL, NULL, NULL,
                    &dwBytes );
```

```
      // Allocate memory for the buffer.
      pszBuffer = (char *) HeapAlloc( GetProcessHeap(),
                                      HEAP_ZERO_MEMORY,
                                      dwBytes );

      // Get the titles and counters.
      RegQueryValueEx( hKeyIndex,
                       "Counters",
                       NULL, NULL,
                       (LPBYTE)pszBuffer,
                       &dwBytes );

      // Find the index value for PROCESS.
      pszTemp = pszBuffer;

      while( i != (int)dwBytes )
      {
         while (*(pszTemp+i) != '\0')
         {
            szIndex[j] = *(pszTemp+i);
            i++;
            j++;
         }
         szIndex[j] = '\0';
         i++;
         j = 0;
         while (*(pszTemp+i) != '\0')
         {
            szObject[j] = *(pszTemp+i);
            i++;
            j++;
         }
         szObject[j] = '\0';
         i++;
         j = 0;
         if( *(pszTemp+i) == '\0' )
            i++;
         if( strcmp(szObject, pszCounter) == 0 )
            break;
      }

      // Deallocate the memory.
      HeapFree( GetProcessHeap(), 0, (LPVOID)pszBuffer );

      // Close the key.
      RegCloseKey( hKeyIndex );
   }
```

REFERENCES
==========

For more information on working with the performance data, please see one or all of the following references:

- The "Win32 Programmer's Reference."

- The "Windows NT Resource Kit," volume 3.

- The source code for PView that is included in the Win32 SDK.

- The "Windows/MS-DOS Developer's Journal," April 1994.

Additional reference words: 3.10 3.50 file name
KBCategory: kbprg
KBSubcategory: BseMisc

Getting the MAC Address for an Ethernet Adapter

PSS ID Number: Q118623

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

To get the Media Access Control (MAC) address for an ethernet adapter programmatically, you can use NetBIOS if your card is bound to NetBIOS. Use the Netbios() NCBASTAT command and provide a "*" as the name in the NCB.ncb_CallName field. This is demonstrated in the sample code below.

With the NetBEUI and IPX transports, the same information can be obtained at a command prompt by using:

```
net config workstation
```

The ID given is the MAC address.

Sample Code

```
#include <windows.h>
#include <wincon.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

typedef struct _ASTAT_
{
   ADAPTER_STATUS adapt;
   NAME_BUFFER    NameBuff [30];
}ASTAT, * PASTAT;

ASTAT Adapter;

void main (void)
{
   NCB Ncb;
   UCHAR uRetCode;
   char NetName[50];

   memset( &Ncb, 0, sizeof(Ncb) );
   Ncb.ncb_command = NCBRESET;
   Ncb.ncb_lana_num = 0;

   uRetCode = Netbios( &Ncb );
   printf( "The NCBRESET return code is: 0x%x \n", uRetCode );

   memset( &Ncb, 0, sizeof (Ncb) );
   Ncb.ncb_command = NCBASTAT;
   Ncb.ncb_lana_num = 0;
```

```
      strcpy( Ncb.ncb_callname,  "*               " );
      Ncb.ncb_buffer = (char *) &Adapter;
      Ncb.ncb_length = sizeof(Adapter);

      uRetCode = Netbios( &Ncb );
      printf( "The NCBASTAT return code is: 0x%x \n", uRetCode );
      if ( uRetCode == 0 )
      {
         printf( "The Ethernet Number is: %02x%02x%02x%02x%02x%02x\n",
                  Adapter.adapt.adapter_address[0],
                  Adapter.adapt.adapter_address[1],
                  Adapter.adapt.adapter_address[2],
                  Adapter.adapt.adapter_address[3],
                  Adapter.adapt.adapter_address[4],
                  Adapter.adapt.adapter_address[5] );
      }
   }
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork
KBSubcategory: NtwkNetbios

Getting the Net Time on a Domain

PSS ID Number: Q98722

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

When trying to execute

```
net time /domain:egdomain /set
```

you may get a message saying the account is not known or the password is invalid. This can happen if you are logged on using an account whose name is spelled "Administrator", but the account is a different Administrator account than the one on the domain controller. For example, if you are logged on as EGMACHINE\Administrator, and attempt

```
net time /domain:egdomain /set
```

you will get an error message because EGMACHINE\Administrator is not the same account as EGDOMAIN\Administrator.

The solution is to log off EGMACHINE, log back on as EGMACHINE\PowerUsr1, then execute the command. Note that a privilege is needed to set the time on a machine. In the previous example, the account, EGMACHINE\PowerUsr1, was used to remind us that power users have the needed privilege.

MORE INFORMATION
================

When running Windows NT while logged on to a domain, doing a NET TIME without the /DOMAIN parameter, as mentioned above, probably will not yield the desired results. However, because you are logged on to a domain, you can do

```
net time /domain /set
```

and a domain controller from the domain you are logged on to will be used. In other words, if you are logged on to a domain, the /DOMAIN parameter is necessary, but the actual domain name can optionally be left to default to the domain you're currently participating in. If your machine is joined to the a domain, that domain will be the default domain for NET TIME /DOMAIN.

If you are trying to get the time from EGDOMAIN and have done a prior

```
net use \\egdomain\ipc$ /user:username
```

where username can be either a legitimate user name or domain name\user

name pair, or anything that will use the guest access), then the net time will use the existing connection to the IPC$ share, using the different user name.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Getting the WinMain() lpCmdLine in Unicode

PSS ID Number: Q90912

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

The prototype for WinMain() is as follows:

```
   int PASCAL WinMain(
          HANDLE hInstance,
          HANDLE hPrevInstance,
          LPSTR lpCmdLine,
          int nCmdShow );
```

The third parameter is an LPSTR, which specifies an ANSI string. WinMain() cannot be defined to accept Unicode input because there is no way for the system to know whether or not the application wants Unicode at the time WinMain() is called; the system knows once the application has registered a window class.

To get the arguments in Unicode, use GetCommandLine().

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrNls

GetWindowRect() Returns TRUE with Desktop Window Handle

PSS ID Number: Q129598

The information in this article applies to:

- Microsoft Win32s, version 1.2

Under Windows NT and Windows 95, GetWindowRect() returns FALSE (to indicate failure) if the desktop window handle is used. Under Win32s, GetWindowRect() returns TRUE if the desktop window handle is used, however, the RECT structure is not correctly filled in.

On Win32s, GetWindowRect() returns TRUE unconditionally because the 16-bit Windows GetWindowRect() has no return value. Therefore, before calling GetWindowRect() on Win32s, you should first check that the window handle is not the desktop window handle.

Additional reference words: 1.20 GetDesktopWindow HWND_DESKTOP
KBCategory: kbprg
KBSubcategory: W32s

GLLT.EXE: SAMPLE: Demonstrates Simple Lighting in OpenGL

PSS ID Number: Q152001

The information in this article applies to:

- Microsoft Win32 Software Development Kit for:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows NT version 4.0 (beta)
  - Microsoft Windows 95 version 4.0

SUMMARY

The GLLight sample provides a demonstration of how the various light settings effect an OpenGL scene. The initial scene is simply a single white sphere with a single blue light (GL_LIGHT0) shining on it.

You can modify all of the properties of that light as well as the light model characteristics for the scene to observe the visual effect. GLLight does not use multiple lights and does not allow you to change the material properties of the sphere.

The GLLight sample is included in GLLT.EXE, a self-extracting file, which can be found on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile GLLT.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get GLLT.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and download GLLT.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download GLLT.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

MORE INFORMATION

GLLight allows you to experiment with all of the OpenGL lighting capabilities for a single-light scene quickly and easily. These characteristics fall into the following categories seen in the "Settings" dialog box:

- Characteristics of the Light Source (GL_LIGHT0):
  - Position
  - Intensity (ambient, diffuse, specular)
  - Spotlight Properties (exponent, cutoff)
  - Attenuation (constant, linear, quadratic)

- Characteristics of the Lighting Model:
  - Global ambient light
  - Viewpoint (local or infinite)
  - Lighting (one-sided or two-sided)

The initial settings for these characteristics follow the defaults for GL_LIGHT0 with two exceptions: the Position is moved back so as to light the sphere; and the Diffuse Intensity is set to blue.

The material properties of the sphere are left as defaults except for a white Specular reflectance and a medium Shininess setting. You cannot modify the material properties through the UI in this sample.

To demonstrate the effects of one-sided vs. two-sided lighting, there is an option to place a clipping plane through the sphere in order to uncover the inside of it. To see the difference between the inside and outside more clearly, you should move the light some distance along the x or y axes.

REFERENCES

For more information on the above settings and OpenGL in general, please read:

-Neider, Jackie, Tom Davis, and Mason Woo. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1. Reading, MA: Addison-Wesley, 1993. ISBN 0-201-63274-8. (This book is also known as the "Red Book".)

For information on Material Properties in OpenGL, please see: The GLBMP Sample "Demonstration of OpenGL Material Property and Printing" in the Microsoft Software Library, and the "Red Book" mentioned above.

Additional reference words: 3.51 4.00 gllt.exe opengl
KBCategory: kbgraphic kbfile
KBSubcategory: GdiOpenGL

Global Classes in Win32

PSS ID Number: Q80382

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Under 16-bit Windows, when an application wants to check whether or not a window class has been previously registered in the system, it typically checks hPrevInstance. Under 32-bit Windows and Windows NT, hPreviousInstance is always FALSE, because a class definition is not available outside the process context of the process that registers it. Thus, code that checks hPreviousInstance will always register the window class.

MORE INFORMATION

Under 32-bit Windows and Windows NT, a style of CS_GLOBALCLASS indicates that the class is available to every DLL in the process, not every application and DLL in the system, as it does in Windows 3.1.

To have a class registered for every process in the system under Windows NT:

1. Register the class in a DLL.

2. Use a style of CS_GLOBALCLASS.

3. List the DLL in the following registry key.

```
HKEY_LOCAL_MACHINE\S OFTWARE\
   Microsoft\
   Windows NT\
   CurrentVersion\
   Windows\
   AppInit_DLLs
```

This will force the DLL to be loaded into every process in the system, thereby registering the class in each and every process.

NOTE: This technique does not work under Windows 95.

For more information, please see "Window Classes in Win32," which is available on the MSDN CD, starting April 1994.

Additional reference words: 3.10 3.50 4.00 95

KBCategory: kbui
KBSubcategory: UsrCls

GLPRT: Sample App Demonstrates Printing in OpenGL

PSS ID Number: Q149769

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), for Windows, version 4.0 (beta)

SUMMARY
=======

Prior to Microsoft Windows NT version 4.0, OpenGL applications that needed to print a rendered scene had to render their scene onto a DIB Section and then print that DIB Section. With Windows NT 4.0, OpenGL applications can now render directly to a printer DC provided the printer is backed by a metafile spooler.

The GLPrint sample included in GLPRT.EXE shows how to set up a printer DC, pixel format descriptor, and viewport in order to render an OpenGL scene to a printer.

You can find GLPRT.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile GLPRT.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get GLPRT.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and download GLPRT.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download GLPRT.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

MORE INFORMATION
================

The GLPrint sample renders a lit-cube to the window. From the File menu, choose Print, and a dialog box appears. When you choose a printer, the scene prints provided that the all of the following conditions are met:

- The print spooler is located on a Windows NT 4.0 (or higher) machine.

- The printer is backed by a metafile spooler.

- EMF spooling is turned on for the printer. Some printers, such as PostScript printers, have EMF spooling turned off by default.

OpenGL graphics are printed in bands in order to conserve memory. Note that this can significantly increase the time it takes to print your scene.

Additional reference words: 4.00 glprint.exe
KBCategory: kbgraphic kbfile kbhowto kbcode
KBSubcategory: GdiOpenGL

Graphics as Jumps in Windows Help Files

PSS ID Number: Q67884

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

One of the things that reduces the training required to learn Windows is the presence of consistent graphical help. You can create Help files that use graphic images as context jumps or as glossary pop-ups.

MORE INFORMATION

To use a graphic as a context jump, perform the following steps:

1. Create the graphic in your favorite graphics editor. The typical Windows developer has at least two tools available for this purpose:

   a. Windows Paintbrush, included with the retail release of Windows

   b. SDKPaint, provided with the Windows SDK

   A Note to Paintbrush Users

   When an image is saved from Paintbrush, by default, the entire "canvas" is saved to disk, not just the image created. To work around this potential problem, after the image is complete, use the square outline tool (at the upper right of the tool bar) to select the image. If the image is not square, select the smallest bounding rectangle, leaving a border if desired. From the Edit menu, choose Copy To, and fill in the name of the file in which to save the selection.

2. Edit the file that contains the Help text. This is often called the .RTF file.

3. Place the cursor where the graphic should occur and turn on the strikethrough character format feature of the text editor. Word for Windows users should use double underline character formatting instead.

4. Insert one of the following text strings:

   {bmc <filename>}
   -or-

{bml <filename>}
-or-
{bmr <filename>}

This text should be struck through (or double underlined).

5. Turn off the strikethrough (or double underline) character format and turn on hidden text character formatting.

6. Type the context string for the jump destination.

7. Turn off hidden text character formatting.

8. Save the .RTF file.

9. Edit the .HPJ file. There must be a [BITMAPS] section in this file, and that section must include the name of the bitmap used above.

When the Help file is built, clicking the graphic with the mouse will cause Help to change to the specified context.

If the graphic has a name or other short text description, we recommend that the text also be coded as a context jump. This way, the user can click on either the graphic or the text to perform the jump. The text also provides a means for users without a mouse to perform the jump.

More information on Windows Help files is in Chapters 15 through 19 of the "Microsoft Windows Software Development Kit Tools" manual. Chapter 17 discusses context jumps, glossary pop-ups, and inserting graphics into the Help file by reference.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsHlp

Graying the Text of a Button or Static Text Control

PSS ID Number: Q39480

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

A control is a child window that is responsible for processing keyboard, mouse, focus, and activation messages, among others. A control paints itself and processes text strings. The color of the text in a button or static text control is automatically changed to gray when the control is disabled with the EnableWindow function. However, If an application subclasses a control to process WM_PAINT messages for the control, the application can use the GrayString function to change the text color.

Additional reference words: 3.00 3.10 3.50 4.00 95 grey
KBCategory: kbui
KBSubcategory: UsrCtl

Handling COMMDLG File Open Network Button Under Win32s

PSS ID Number: Q117825

The information in this article applies to:

- Microsoft Win32s versions 1.1, 1.15, and 1.2

When you run a Win32-based application under Win32s on top of Windows 3.1 without a network, the File Open common dialog box still has a Network button.

Under Windows NT, the File Open common dialog box that appears when you call GetOpenFileName() has a Network button only when a network is present. The API (application programming interface) will either use the Network button in the dialog box template if it exists or dynamically add the button if it is not in the template and there is a network present. If there is no network present but the template contains a Network button, the button will be hidden.

The dialog box template that is included with the Windows Software Development Kit (SDK) does not have a Network button because Windows does not include a network. When the 16-bit Windows-based application is executed under Windows for Workgroups or under Windows NT, the Network button is dynamically added, because these operating systems have built-in networking.

The template that is included with the Win32 SDK has a Network button. If the Win32-based application is run under a non-networked Windows 3.1, the Network button is shown; however it is nonfunctional because there is no network. This happens because the COMMDLG.DLL that provides Windows 3.1 with the common dialog boxes does not recognize networks. Therefore, GetOpenFileName() does not remove the Network button from the template if there is no network.

One solution is to leave the button in the template, determine when you are running under Win32s, and include OFN_NONETWORKBUTTON in the OPENFILENAME structure when Win32s is present but there is no network present. Define a hook function that during WM_INITDIALOG checks the Flags field of the OPENFILENAME struct that lParam is pointing to. If OFN_NONETWORKBUTTON is used, call ShowWindow(GetDlgItem(hWnd, psh14), SW_HIDE).

Alternatively, if your application will most likely run on networked Windows 3.1 machines, you can install the COMMDLG.DLL that ships with Windows for Workgroups 3.11 on all machines, because it is a redistributable dynamic-link library (DLL). This DLL checks to see if a network is present and removes the Network button for you if it is not.

Additional reference words: 1.10 1.15 1.20
KBCategory: kbprg
KBSubcategory: W32s

Handling WM_CANCELMODE in a Custom Control

PSS ID Number: Q74548

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In the Microsoft Windows graphical environment, the WM_CANCELMODE message informs a window that it should cancel any internal state. This message is sent to the window with the focus when a dialog box or a message box is displayed, giving the window the opportunity to cancel states such as mouse capture.

When a control has the focus, it receives a WM_CANCELMODE message when the EnableWindow function disables the control or when a dialog box or a message box is displayed. When a control receives this message, it should cancel modes, such as mouse capture, and delete any timers it has created. A control must cancel these modes because an application may use a notification from the control to display a dialog box or a message box.

The DefWindowProc function processes WM_CANCELMODE by calling the ReleaseCapture function, which cancels the mouse capture for whatever window has the capture. The DefWindowProc function does not cancel any other modes.

MORE INFORMATION

For example, consider a miniature scroll bar custom control that, when it receives a mouse click, sets the mouse capture, creates a timer to provide for repeated scrolling, and sends a WM_VSCROLL message to its parent application. The timer is used to send WM_VSCROLL messages periodically to the parent when the mouse button is held down and the mouse is over the control.

If the application displays a dialog box in response to the WM_VSCROLL message, the control receives a WM_CANCELMODE message, at which time it should kill its timer and release the mouse capture. If the WM_CANCELMODE message is simply passed to the DefWindowProc function, only the mouse capture is released; the timer remains active. When the dialog box is closed, the control immediately sends the parent another WM_VSCROLL message, causing it to display the dialog box again.

Additional reference words: 3.00 3.10 3.50 4.00 95

KBCategory: kbui
KBSubcategory: UsrCtl

Height and Width Limitations with Windows SDK Font Editor

PSS ID Number: Q69081

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

The Microsoft Windows Font Editor (FONTEDIT.EXE) does not allow creation of fonts in which the height or width of the font is greater than 64 pixels. This corresponds to a font-file size of approximately 115K.

This is a limitation of the Font Editor, and not of the fonts themselves. There is no limit to the size of the font file in the font format. Font files created in the Windows version 2.0 font format cannot be larger than 64K.

The Font Editor displays a message box containing one of these two errors if the entered Font Size Character Pixel Width or Character Pixel Height is larger than 64:

   Fixed/maximum width must be a number from 1-64

   Font height must be a number from 1-64 pixels

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsFnt

Help Fonts Must Use ANSI Character Set

PSS ID Number: Q92422

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

=======

When creating a help file, it is often necessary to access special characters available in certain fonts. However, when the Windows Help engine tries to display the special character, it may not appear correctly.

MORE INFORMATION

================

Fonts used to create RTF source files for the Windows Help engine must use the ANSI character set. Fonts that use character sets other than ANSI may not display properly in the Help engine. The only exception to this is the Symbol font, which is also supported in Windows Help version 3.1.

To determine whether a particular font uses the ANSI character set, run the FONTEST sample included with the Windows Software Development Kit (SDK). From the Font menu, choose the Choose Font option. In the Font dialog box, select the font for which you would like to determine the character set. If the font is an ANSI font, then the tmCharSet value displayed in the main window will be zero.

Additional reference words: 3.10 3.50 4.00 95 HC HCP

KBCategory: kbtool

KBSubcategory: TlsHlp

High-Precision Timing Under Windows, Windows NT, & Windows 95

PSS ID Number: Q148404

The information in this article applies to:

- Microsoft Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK) for:
  - Windows NT versions 3.5 and 3.51
  - Windows 95

SUMMARY

Windows 3.1 developers who require high precision timers can use the multimedia timers, which are accurate to 10 milliseconds (ms) in most cases. However, in the multithreaded environments of Windows NT and Windows 95, the resolution and accuracy of the multimedia timers may often be reduced due to the preemptive nature of the operating systems.

MORE INFORMATION

In Windows 3.1, multimedia timers are serviced by an interrupt service routine (ISR) which is tied to the computer's clock. When a tick on the clock occurs, Windows suspends the running program and services the tick. Should your application use multimedia timers (by using TimeSetEvent() and any of the CALLBACK flags), Windows executes your callback function before returning control to the suspended program. Servicing your callback function during the hardware timer tick provides high precision timing because your callback is being executed within the timer interrupt, and interrupt routines cannot, themselves, be interrupted.

The multithreaded environments of Windows NT and Windows 95 employ CPU scheduling techniques to improve processor utilization and smooth multitasking. The portions of code, called threads, within the system are organized into queues and assigned priorities. Threads of code required by the operating system are assigned high priorities. The thread scheduler allows a thread a predetermined amount of execution time (quantum). Then it evicts the thread from the processor until all threads at the specific priority level have been serviced. The process continues at the next, lower, priority level. The timer-servicing thread, like applications, is subject to CPU scheduling and eviction regardless of the thread priority. Given that the number of threads ready for execution changes frequently, the period between timer thread executions can vary dramatically.

For example, say five time-critical threads are ready for execution, and the quantum is q milliseconds. If the timer thread is fifth, four threads execute before the timer thread is executed; q milliseconds x 4. This means that if q is 25 milliseconds, 100 milliseconds will have elapsed before the fifth thread is executed. Boosting the priority of the thread that created the timer risks introducing instability in the system by starving the time-sensitive, system-critical threads.

Additional reference words: 4.00 3.50 multimedia resolution flat generic
KBCategory: kbmm kbprg kbhowto
KBSubcategory: MMTimer

Hooking Console Applications and the Desktop

PSS ID Number: Q108232

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Under Windows NT, system hooks are limited in two situations: hooking console windows and hooking the desktop.

Because of the current design of the console and the fact that its user interface runs in the Win32 server, Windows NT does not support hook calls in the context of console applications. Thus, if application A sets a system-wide input hook and text is typed in a console window, application A's input hook will not be called. The same is true for every type of Windows hook except for journal record and journal playback hooks.

Hooking a console application will be enabled in Windows NT 3.51.

MORE INFORMATION

Windows NT supports journaling by forcing the console input thread to communicate with the application that set the hook. In the case of a console, the call to the hook functions are run in the context of the application that installed the hook. This forces Windows NT to synchronously talk to this process in order for it to work; if this process is busy or blocked (as it is when it is sitting at a breakpoint), the console thread is hung.

If console applications were typical Win32-based applications, then this would not be a problem. A design change such as this would require that each console take an extra thread just to process input. This was not acceptable to the designers, and therefore console applications are not implemented in the same way that other Win32-based applications are implemented.

Similarly, if Windows NT allowed other hooks to freely hook any process, then these processes could enter a hanging state as well. The reason that journaling is allowed to hook consoles is that journaling already requires synchronization between all processes in the system, and a mechanism to disengage the journaling process (via the CTRL+ESC, ALT+ESC and CTRL+ALT+DEL keystroke combinations) is provided to prevent hanging the system message queue.

For similar reasons, 16-bit Windows-based applications cannot hook Win32-based applications under Windows NT.

The issues above apply equally well to hooking the desktop. If an application were allowed to hook the desktop, it could potentially hang it. This is completely unacceptable and violates one of the design principles of Windows NT: no application should be allowed to bring down the system or hang the user interface.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrHks

Host Name May Map to Multiple IP Addresses

PSS ID Number: Q110703

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

The sockets function gethostname() returns a string that identifies the name of the local host. Each host has only one "official" name, regardless of how many IP addresses it has, but there may be several "aliases" for the host.

In TCP/IP, there is not a one-to-one mapping between host name and IP address. The mapping is one-to-many: one host name can have multiple IP addresses.

The sockets function getsockname() returns the sockaddr that the socket is bound to.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork
KBSubcategory: NtwkWinsock

Hot Versus Warm Links in a DDEML Server Application

PSS ID Number: Q108927

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In message-based dynamic-data exchange (DDE), a server application can readily distinguish between a hot and a warm link as soon as it receives a request for an advise loop, via the WM_DDE_ADVISE message. This allows the server to send the appropriate value in the data handle (for example, a NULL or a valid data handle) to the client application whenever data changes. In DDEML, there is no way to distinguish between these two links when the server receives a request for an advise transaction.

MORE INFORMATION

Two applications engaged in a DDE conversation may establish one or more links (or advise loops) so that the server application sends periodic updates about the linked item/s to the client, typically when that particular data item's value changes.

In a hot advise loop, the server immediately sends a data handle to the changed data item value. In a warm advise loop, however, the server just notifies the client that the data item value has changed, but does not send the data handle until the client explicitly requests it.

In message-based DDE, a client requests the server for an update on an item by posting the WM_DDE_ADVISE message to the server application. Upon receipt of this message, the server application is able to distinguish between a request for a hot advise loop and a warm advise loop via the fDeferUpd bit of the DDEADVISE structure it received in the low-order word of lParam.

A nonzero fDeferUpd value tells the server that it is a WARM advise loop. This instructs the server to send a WM_DDE_DATA message with a NULL data handle whenever the data item changes, and wait for the client to post a WM_DDE_REQUEST before it sends the handle to the updated data.

A zero fDeferUpd value, however, indicates a HOT advise loop, which then tells the server to send a WM_DDE_DATA message with the valid data handle to the changed data item.

In DDEML, a client requests the server for a hot advise loop via the

XTYP_ADVSTART transaction type in a call to the DdeClientTransaction() function. To request a warm advise loop, the client specifies an XTYP_ADVSTART transaction or'ed with the XTYPF_NODATA flag. In both cases, the DDEML passes the same XTYP_ADVSTART to the server callback function, with no particular flags set, leaving the server with no means to distinguish between a hot or warm advise request.

Note that DDEML internally remembers the type of advise loop established.

Once an advise loop is established, the server application calls the DdePostAdvise() function whenever the value of the data item changes. In a hot advise loop, this causes the DDEML to send the server an XTYP_ADVREQ transaction to its callback function, where the server then returns a data handle to the changed data item. The DDEML then sends the XTYP_ADVDATA transaction to the client's callback function with the data handle.

In a warm advise loop, an XTYP_ADVREQ transaction is not sent to the server's callback function when the data item changes on a call to DdePostAdvise(). DDEML takes care of sending the XTYP_ADVDATA transaction directly to the client's callback function, with the data handle set to NULL. The server application does not send the handle to the changed data item until the client issues an XTYP_REQUEST transaction to obtain this data handle.

Because the type of advise loop (hot versus warm) is not known to the server application, a good rule of thumb in writing server applications that support advise loops is to return a data handle in response to both the XTYP_ADVREQ and the XTYP_REQUEST transactions. This guarantees that a data handle is returned for both hot and warm advise loops.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

How CREATOR_OWNER and CREATOR_GROUP Affect Security

PSS ID Number: Q126629

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SUMMARY

=======

This article discusses the CREATOR_OWNER and CREATOR_GROUP security identifiers (SID) and how they affect security.

MORE INFORMATION

================

When logged on, each user is represented by a token object. This token contains all the SIDs comprising your security context. Tokens identify one of those SIDs as a default owner for any new objects the user creates, such as files, processes, events, and so forth. Typically, this is the user's account (<domain>\<username>). For an administrator however, the default owner is set to be the local group "Administrators," rather than the individual's user account.

Each token also identifies a primary group for the user. This group does not necessarily have to be one the user is a member of (although it is by default) and it does not determine the objects a user has access to (that is, it isn't used in access validation decisions). However, by default it is assigned as the primary group of any objects the user creates. For the most part, the primary group is required simply for POSIX compatibility, but the primary group does play a role in object creation.

When a new object is created, the security system has the task of assigning protection to the new object. The system follows this process:

1. Assign the new object any protection explicitly passed in by the object creator.

2. Otherwise, assign the new object any inheritable protection from the container the object is created in.

3. Otherwise, assign the new object any protection explicitly passed in by the object creator, but marked as "default."

4. Otherwise, if the caller's token has a default DACL, that will be assigned to the new object.

5. Otherwise, no protection is assigned to the new object.

In step 2, if the parent container has inheritable access-control entries (ACE), those are used to generate protection for the new object. In this

case, each ACE is evaluated to see if it should be copied to the new object's protection. Usually, when an ACE is copied, the SID within that ACE is copied as is. The two exceptions to this rule are when CREATOR_OWNER and CREATOR_GROUP are encountered. In this case, the SID is replaced with the caller's default owner SID or primary group SID.

By default, users logging on to Windows NT are given a primary group of "Domain Users" (when logging on to a Windows NT Server) or the group called "None" (when logging onto a Windows NT Workstation system). Therefore, when you create an object in a container that has an inheritable ACE with the CREATOR_GROUP SID, you will likely end up with an ACE granting Domain Users some access. This may not be what you intended.

Additional reference words: 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

How Database WinSock APIs Are Implemented in Windows NT 3.5

PSS ID Number: Q130024

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.5 and 3.51

SUMMARY
=======

This article describes the ways in which various WinSock database APIs are implemented through the Windows NT versions 3.5 and 3.51 implementation of the WinSock DLL. The article covers the following WinSock database APIs: gethostbyname(), gethostbyaddr(), getprotobyname(), getprotobynumber(), getservbyname(), and getservebynumber().

MORE INFORMATION
================

Following are the steps taken by each API. In a case where more than one step may be taken to resolve the requested information, the process is not carried to the next step if the information is resolved in the current step.

gethostbyname():

1. Check the HOSTS file at %SystemRoot%\System32\DRIVERS\ETC.
2. Do a DNS query if the DNS server is configured for name resolution.
3. Query one or more WINS servers.

gethostbyaddr():

1. Check the HOSTENT cache.
2. Check the HOSTS file at %SystemRoot%\System32\DRIVERS\ETC.
3. Do a DNS query if the DNS server is configured for name resolution.
4. Do an additional NetBIOS remote adapter status to an IP address for its NetBIOS name table. This step is specific only to the Windows NT version 3.51 implementation.

getprotobyname() and getprotobynumber():

1. Check the PROTOCOL file at %SystemRoot%\System32\DRIVERS\ETC.

getservbyname() and getservebynumber():

1. Check the SERVICES files at %SystemRoot%\System32\DRIVERS\ETC.

Additional reference words: 3.50 3.51
KBCategory: kbnetwork kbnetwork
KBSubcategory: NtwkWinsock

How HEAPSIZE/STACKSIZE Commit > Reserve Affects Execution

PSS ID Number: Q89296

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The syntax for the module-definition statements HEAPSIZE and STACKSIZE is as follows

```
HEAPSIZE  [reserve] [,commit]
STACKSIZE [reserve] [,commit]
```

The remarks for HEAPSIZE and STACKSIZE on page 62 of the "Tools User's Guide" manual that comes with the Win32 SDK state the following:

> When commit is less than reserve, memory demands are reduced but execution time is slower.

By default, commit is less than reserve.

The reason that execution time is slower (and it is actually only fractionally slower), is that the system sets up guard pages and could have to process guard page faults.

MORE INFORMATION

If the committed memory is less than the reserved memory, the system sets up guard page(s) around the heap or stack. When the heap or stack grows big enough, the guard pages start accessing outside the committed area. This causes a guard page fault, which tells the system to map in another page. The application continues to run as if you had originally had the new page committed.

If the committed memory is greater than the reserve, no guard pages are created and the program faults if it goes outside the committed memory area.

Experimenting with the commit versus reserve numbers may result in a combination that would produce noticeable results, but for most applications, this difference is probably not noticeable. The potential benefits do not warrant significant experimentation.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

How Keyboard Data Gets Translated

PSS ID Number: Q104316

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Keyboard input is acquired by the keyboard driver, which in turn produces a scan code. This scan code is passed on to the locale-specific Win32 subsystem keyboard driver. This locale-specific driver then converts the scan code to a virtual key and a Unicode character. The Win32 subsystem then passes on this information to the application.

All messages in the Win32 application programming interface (API) that present textual information to a window procedure depend upon how the window registered its class. For example, if RegisterClassW() was called, then Unicode is presented; if RegisterClassA() was called, then ANSI is presented. The conversion of the text is handled by the Window Manager. This allows an ANSI application to send textual information to a Unicode application.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrInp

How the Service Control Manager Manages Passwords

PSS ID Number: Q149641

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SUMMARY

=======

Services have the ability to run in the security context of an ordinary user account. In order for a service to do this, the Service Control Manager needs the name and password of the account in which it is to run.

The Service Control Manager keeps two copies of a user account's password, the current password and the backup password. The first time you install a service, the password given to the Service Control Manager is stored as the current password and the backup password is not initialized.

When the Service Control Manager attempts to log the service in the security context of the user account, it uses the current password. If the current password is successful, it is also saved as the backup password. If the user account's password is modified with ChangeServiceConfig() or the Services control panel applet, the modified password is stored as the current password and the previous current password is stored as the backup password. When an attempt is made to start the service, the Service Control Manager uses the current password. If the current password fails, the Service Control Manager uses the backup password. If the backup password is successful, it becomes the current password.

MORE INFORMATION

================

If the Service Control Manager cannot start the service in the security context of the user account with the current password, the following event log entry is logged into the system log:

```
Event ID    = 7013
Source      = Service Control Manager
Type        = Error
Description = Logon attempt with current password failed with the
              following error: Logon failure: unknown user name or
              bad password.
```

If the Service Control Manager cannot start the service in the security context of the user account with the backup password, the following event log entry is logged into the system log:

```
Event ID    = 7014
Source      = Service Control Manager
Type        = Error
Description = Second logon attempt with old password also failed with
              the following error: Logon failure: unknown user name or
```

bad password.

Additional reference words:
KBCategory: kbprg
KBSubcategory: bseservice

How to Add a Custom Find Utility to the Start Menu

PSS ID Number: Q135986

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

=======

In Windows 95, you can have a program create a custom Find utility for the Start Menu. For an example, look at how The Microsoft Network adds a custom Find "On the Microsoft Network..." option to Windows 95 when the user signs up on The Microsoft Network.

MORE INFORMATION

================

The first step to registering a custom Find utility is to implement a shell context menu extension that supplies the menu item. The handler's IContextMenu::InvokeCommand() member will be called whenever the user selects the custom Find utility on the Start Menu.

The context menu extension can be registered by adding some information for the handler to the following key in the registry:

```
\HKEY_LOCAL_MACHINE
    \Software
        \Microsoft
            \Windows
                \CurrentVersion
                    \Explorer
                        \FindExtensions
                            \Static
```

Under this key, your program needs to add a key, such as MyFind, for the context menu extension that contains the GUID for the context menu handler. Under MyFind, your program needs to create a second key "0" that contains the text of the menu item you want to include. Finally, under the "0" key, you need to create a third key "DefaultIcon" that contains the name of a .dll file that contains the icon you want displayed on the menu and the index of the icon in that .dll file.

To summarize, you need to add the following under the ..\FindExtensions\Static key in the registry:

```
\MyFind = <GUID>
    \0 = <Text for the menu item>
        \DefaultIcon = <path to dll, index of icon in dll>
```

The context menu handler must also be registered normally as documented in

the Win32 SDK.

REFERENCES
==========

You can find additional information on writing context menu handlers in:

 - The Win32 SDK's "Programmer's Guide to Windows 95" on MSDN.

 - Nancy Clut's book "Programming the Windows 95 UI."

 - Jeff Prosise's March 1995 MSJ article "Writing Windows 95 Shell
   Extensions."

Additional reference words: 4.00 Windows 95
KBCategory: kbui
KBSubcategory: UsrShell

How to Add a Custom Tab to the Help Topics Dialog Box

PSS ID Number: Q139834

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API)
  included with:

  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

This article discusses how to add a custom tab to the Help Topics dialog box. This option is only available in version 4.0 or later of the Help Compiler for Windows.

MORE INFORMATION
================

Create a Dialog Resource

To add a custom tab to the WinHelp Help Topics dialog box, you must do three things:

1. Create a dialog template resource to display in the custom help tab. The dialog template that you create will be displayed over the top of the existing Help Tab dialog box. You do not need to worry about the size of the dialog frame; WinHelp will automatically resize it at display time to ensure that it fits.

2. Ensure that the dialog template that you create has no borders and includes the WS_CHILD and the DS_CONTROL styles. DS_CONTROL is a new style that ensures that the dialog box will get the focus when the tab is activated. DS_CONTROL is defined in Winuser.h file included with the Win32 SDK. The definition is as follows:

   ```
   #define DS_CONTROL          0x0400L
   ```

3. Make the dialog box itself visible. On this last point, Help (Hcw.hlp) incorrectly suggests that the dialog box should be invisible. This is an error. If you do not ensure that the dialog box is visible, you will not be able to see it when you click on the custom help tab.

Write a DLL to Support the Dialog Resource

Once you have created the dialog resource, you must write a DLL to support it. In addition to including the dialog box procedure, the DLL must export the following function, prototyped as follows:

```
   HWND WINAPI OpenTabDialog(HWND, DWORD, DWORD)

This function will be called automatically by WinHelp when the custom tab
is clicked; at which point, WinHelp passes the window handle of the tab
dialog box as the first parameter. This allows you to call CreateDialog()
from within the function. The remaining two parameters are reserved for
future use and can be ignored.

Example DLL
-----------

Here is an example of what this DLL might look like:

#include <windows.h>
#include "resource.h"

HINSTANCE hinst;  // Used by OpenTabDialog()

/* DllMain exists solely for the purpose of obtaining the DLL instance
 * and storing it in a global variable used to call CreateDialog().
 */
BOOL WINAPI DllMain (HINSTANCE hinstDll, DWORD fdwReason,
  LPVOID lpReserved)
{

     switch (fdwReason){

          case DLL_PROCESS_ATTACH:
               hinst = hinstDll;
               break;
     }

     return (TRUE);
}

BOOL WINAPI  DialogProc (HWND hDlg, UINT message, UINT wParam,
  long lParam)
{
     . . .

     return FALSE;
}

/* This procedure is called by WINHLP32. Winhelp passes in its hwnd
 * so you can call Create Dialog. The instance handle for the DLL is
 * stored in a global variable that is initialized in LibMain.

HWND WINAPI OpenTabDialog(HWND hwnd, DWORD dwReserved1, DWORD dwReserved2){


     return (CreateDialog (hinst, MAKEINTRESOURCE(IDD_DIALOG1), hwnd,
  DialogProc));

}
```

Modify the Help Project (.hpj) File

Once the DLL has been written, you must modify the Help project file (.hpj file) to include the custom tab and link the DLL to it. If you are using HCW, open your Contents file (.cnt file). (If you have not created one, you must do so in order to use your custom tab.) In the Contents dialog box, follow these steps:

1. Click the Tabs button to bring up the Custom Tabs dialog box.

2. Click the Add button to bring up the Add Tab dialog box.

3. Enter the name to appear on the tab in the Tab Name edit box.

4. Enter the name of your custom .dll file (and optionally the path) in the DLL File Name edit box.

5. Click OK twice to exit the dialog boxes.

6. On the File menu, click Save to save your changes.

7. Compile the .hpj file. When you use your help file, the custom tab should now be available to you.

If you do not wish to work with HCW, you can open your Contents file in a text editor, and add the following line:

```
:Tab <tab name>=<path to DLL>

:Tab mytab=c:\helptab\debug\helptab.dll
```

Once you make this change, you will need to save the file and recompile your .hpj file.

Additional reference words: 4.00
KBCategory: kbtool kbdocerr kbhowto kbcode
KBSubcategory: tlshlp

How to Add a Network Printer Connection

PSS ID Number: Q147202

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:

    - Windows NT version 3.51
    - Windows 95

SUMMARY
=======

When writing a Win32 application that installs a network printer connection, you must take into consideration differences in printing if the application will be run under both Windows 95 and Windows NT. One important difference is that Windows NT may download appropriate drivers from the print server while under Windows 95, you need to add the printer driver files to the system with the AddPrinterDriver() function.

MORE INFORMATION
================

This process should take two separate code paths depending on whether you are running Windows NT or Windows 95. In Windows 95, you will want to add the printer driver with AddPrinterDriver() and add the printer connection with AddPrinter(). In Windows NT, you need only call the AddPrinterConnection() function.

Steps for Adding a Printer Connection in Windows 95
---

1. Fill out a DRIVER_INFO_2 structure with the appropriate driver information for the network printer.

2. Copy the driver files to your Windows 95 System directory.

3. Call AddPrinterDriver() to add the printer driver to Windows 95.

4. Fill out a PRINTER_INFO_2 structure, and place a pointer to the server and share name in the pPortName field.

5. Call AddPrinter() to add the printer to the system.

Here is an example of how you might do this (assuming you have copied the driver files to the Windows 95 System directory):

```
PRINTER_INFO_2 pi2;
DRIVER_INFO_2 di2;
HANDLE hPrinter;

ZeroMemory(&di2, sizeof(DRIVER_INFO_2));
di2.cVersion = 1024;
```

```
   di2.pName = "HP Laserjet 4Si";
   di2.pEnvironment = "Windows 4.0";
   di2.pDriverPath = "c:\\windows\\system\\hppcl5ms.drv";
   di2.pDataFile   = "c:\\windows\\system\\hppcl5ms.drv";
   di2.pConfigFile = "c:\\windows\\system\\hppcl5ms.drv";
   AddPrinterDriver(NULL, 2, (LPBYTE)&di2);

   ZeroMemory(&pi2, sizeof(PRINTER_INFO_2));
   pi2.pPrinterName = "HP Laserjet 4Si";
   pi2.pPortName = "\\\\server\\print_share";
   pi2.pDriverName = "HP Laserjet 4Si";
   pi2.pPrintProcessor = "WinPrint";
   pi2.pDatatype = "EMF";
   hPrinter = AddPrinter(NULL, 2, (LPBYTE)&pi2);
   ClosePrinter(hPrinter);
```

6. Make sure you close the printer handle returned from AddPrinter, by using the ClosePrinter API.

You can add a printer connection under Windows NT by making this call:

```
   AddPrinterConnection ("\\\\server\\print_share");
```

Additional reference words: kbinf 4.00
KBCategory: kbgraphic kbhowto kbcode
KBSubcategory: GdiPrnDrvrs

How to Add an Access-Allowed ACE to a File

PSS ID Number: Q102102

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

This article explains the process of adding an access-allowed (or access-denied) access control entry (ACE) to a file.

Adding an access-allowed ACE to a file's access control list (ACL) provides a means of granting or denying (using an access-denied ACE) access to the file to a particular user or group. In most cases, the file's ACL will not have enough free space to add an additional ACE, and therefore it is usually necessary to create a new ACL and copy the file's existing ACEs over to it. Once the ACEs are copied over and the access-allowed ACE is also added, the new ACL can be applied to the file's security descriptor (SD). This process is explained in detail in the section below. Sample code is provided at the end of this article.

MORE INFORMATION

At the end of this article is sample code that defines a function named AddAccessRights(), which adds an access-allowed ACE to the specified file allowing the specified access. Steps 1-17 in the comments of the sample code are discussed in detail below:

1. GetUserName() is called to retrieve the name of the currently logged in user. The user name is stored in plszUserName[] array.

2. LookupAccountName() is called to obtain the SID of the user returned by GetUserName() in step 1. The resulting SID is stored in the UserSID variable and will be used later in the AddAccessAllowedACE() application programming interface (API) call. The LookupAccountName() API is also providing the user's domain in the plszDomain[] array. Please note that LookupAccountName() returns the SID of the first user or group that matches the name in plszUserName.

3. GetFileSecurity() is used here to obtain a copy of the file's security descriptor (SD). The file's SD is placed into the ucSDbuf variable, which is declared a size of 65536+SECURITY_DESCRIPTOR_MIN_LENGTH for simplicity. This value represents the maximum size of an SD, which ensures the SD will be of sufficient size.

4. Here we initialize the new security descriptor (NewSD variable) by

calling the InitializeSecurityDescriptor() API. Because the SetFileSecurity() API requires that the security item being set is contained in a SD, we create and initialize NewSD.

5. Here GetSecurityDescriptorDacl() retrieves a pointer to the discretionary access control list (DACL) in the SD. The pointer is stored in the pACL variable.

6. GetAclInformation() is called here to obtain size information on the file's DACL in the form of a ACL_SIZE_INFORMATION structure. This information is used when computing the size of the new DACL and when copying ACEs.

7. This statement computes the exact number of bytes to allocate for the new DACL. The AclBytesInUse member represents the number of bytes being used in the file's DACL. We add this number to the size of an ACCESS_ALLOWED_ACE and the size of the user's SID. Subtracting the size of a DWORD is an adjustment required to obtain the exact number of bytes necessary.

8. Here we allocate memory for the new ACL that will ultimately contain the file's existing ACEs plus the access-allowed ACE.

9. In addition to allocating the memory, it is important to initialize the ACL structure as we do here.

10. Here we check the bDaclPresent flag returned by GetSecurityDescriptorDacl() to see if a DACL was present in the file's SD. If a DACL was not present, then we skip the code that copies the file's ACEs to the new DACL.

11. After verifying that there is at least one ACE in the file's DACL (by checking the AceCount member), we begin the loop to copy the individual ACEs to the new DACL.

12. Here we get a pointer to an ACE in the file's DACL by using the GetAce() API.

13. Now we add the ACE to the new DACL. It is important to note that we pass MAXDWORD for the dwStartingAceIndex parameter of AddAce() to ensure the ACE is added to the end of the DACL. The statement ((PACE_HEADER)pTempAce)->AceSize provides the size of the ACE.

14. Now that we have copied all the file's original ACEs over to our new DACL, we add the access-allowed ACE. The dwAccessMask variable will contain the access mask being granted. GENERIC_READ is an example of an access mask.

15. Because the SetFileSecurity() API can set a variety of security information, it takes a pointer to a security descriptor. For this reason, it is necessary to attach our new DACL to a temporary SD. This is done by using the SetSecurityDescriptorDacl() API.

16. Now that we have a SD containing the new DACL for the file, we set the DACL to the file's SD by calling SetFileSecurity(). The

DACL_SECURITY_INFORMATION parameter indicates that we want the DACL in the provided SD applied to the file's SD. Please note that only the file's DACL is set, the other security information in the file's SD remains unchanged.

17. Here we free the memory that was allocated for the new DACL.

The below sample demonstrates the basic steps required to add an access-allowed ACE to a file's DACL. Please note that this same process can be used to add an access-denied ACE to a file's DACL. Because the access-denied ACE should appear before access-allowed ACEs, it is suggested that the call to AddAccessDeniedAce() precede the code that copies the existing ACEs to the new DACL.

Sample Code

```
#define SD_SIZE (65536 + SECURITY_DESCRIPTOR_MIN_LENGTH)

BOOL AddAccessRights(CHAR *pFileName, DWORD dwAcessMask)
{
   // SID variables

   UCHAR          psnuType[2048];
   UCHAR          lpszDomain[2048];
   DWORD          dwDomainLength = 250;
   UCHAR          UserSID[1024];
   DWORD          dwSIDBufSize=1024;

   // User name variables

   UCHAR          lpszUserName[250];
   DWORD          dwUserNameLength = 250;

   // File SD variables

   UCHAR          ucSDbuf[SD_SIZE];
   PSECURITY_DESCRIPTOR pFileSD=(PSECURITY_DESCRIPTOR)ucSDbuf;
   DWORD          dwSDLengthNeeded;

   // ACL variables

   PACL           pACL;
   BOOL           bDaclPresent;
   BOOL           bDaclDefaulted;
   ACL_SIZE_INFORMATION AclInfo;

   // New ACL variables

   PACL           pNewACL;
   DWORD          dwNewACLSize;

   // New SD variables

   UCHAR               NewSD[SECURITY_DESCRIPTOR_MIN_LENGTH];
```

```
PSECURITY_DESCRIPTOR psdNewSD=(PSECURITY_DESCRIPTOR)NewSD;

// Temporary ACE

PVOID          pTempAce;
UINT           CurrentAceIndex;

// STEP 1: Get the logged on user name

if(!GetUserName(lpszUserName,&dwUserNameLength))
{
   printf("Error %d:GetUserName\n",GetLastError());
   return(FALSE);
}

// STEP 2: Get SID for current user

if (!LookupAccountName((LPSTR) NULL,
    lpszUserName,
    UserSID,
    &dwSIDBufSize,
    lpszDomain,
    &dwDomainLength,
    (PSID_NAME_USE)psnuType))
{
   printf("Error %d:LookupAccountName\n",GetLastError());
   return(FALSE);
}

// STEP 3: Get security descriptor (SD) for file

if(!GetFileSecurity(pFileName,
              (SECURITY_INFORMATION)(DACL_SECURITY_INFORMATION),
              pFileSD,
              SD_SIZE,
              (LPDWORD)&dwSDLengthNeeded))
{
   printf("Error %d:GetFileSecurity\n",GetLastError());
   return(FALSE);
}

// STEP 4: Initialize new SD

if(!InitializeSecurityDescriptor(psdNewSD,SECURITY_DESCRIPTOR_REVISION))
{
   printf("Error %d:InitializeSecurityDescriptor\n",GetLastError());
   return(FALSE);
}

// STEP 5: Get DACL from SD

if (!GetSecurityDescriptorDacl(pFileSD,
                &bDaclPresent,
                &pACL,
                &bDaclDefaulted))
```

```
{
   printf("Error %d:GetSecurityDescriptorDacl\n",GetLastError());
   return(FALSE);
}

// STEP 6: Get file ACL size information

if(!GetAclInformation(pACL,&AclInfo,sizeof(ACL_SIZE_INFORMATION),
   AclSizeInformation))
{
   printf("Error %d:GetAclInformation\n",GetLastError());
   return(FALSE);
}

// STEP 7: Compute size needed for the new ACL

dwNewACLSize = AclInfo.AclBytesInUse +
               sizeof(ACCESS_ALLOWED_ACE) +
               GetLengthSid(UserSID) - sizeof(DWORD);

// STEP 8: Allocate memory for new ACL

pNewACL = (PACL)LocalAlloc(LPTR, dwNewACLSize);

// STEP 9: Initialize the new ACL

if(!InitializeAcl(pNewACL, dwNewACLSize, ACL_REVISION2))
{
   printf("Error %d:InitializeAcl\n",GetLastError());
   LocalFree((HLOCAL) pNewACL);
   return(FALSE);
}

// STEP 10: If DACL is present, copy it to a new DACL

if(bDaclPresent)  // only copy if DACL was present
{
   // STEP 11: Copy the file's ACEs to our new ACL

   if(AclInfo.AceCount)
   {
      for(CurrentAceIndex = 0; CurrentAceIndex < AclInfo.AceCount;
         CurrentAceIndex++)
      {
         // STEP 12: Get an ACE

         if(!GetAce(pACL,CurrentAceIndex,&pTempAce))
         {
           printf("Error %d: GetAce\n",GetLastError());
           LocalFree((HLOCAL) pNewACL);
           return(FALSE);
         }

          // STEP 13: Add the ACE to the new ACL
```

```
            if(!AddAce(pNewACL, ACL_REVISION, MAXDWORD, pTempAce,
               ((PACE_HEADER)pTempAce)->AceSize))
            {
               printf("Error %d:AddAce\n",GetLastError());
               LocalFree((HLOCAL) pNewACL);
               return(FALSE);
            }
         }
      }
   }

   // STEP 14: Add the access-allowed ACE to the new DACL

   if(!AddAccessAllowedAce(pNewACL,ACL_REVISION2,dwAcessMask, &UserSID))
   {
      printf("Error %d:AddAccessAllowedAce",GetLastError());
      LocalFree((HLOCAL) pNewACL);
      return(FALSE);
   }

   // STEP 15: Set our new DACL to the file SD

   if (!SetSecurityDescriptorDacl(psdNewSD,
                   TRUE,
                   pNewACL,
                   FALSE))
   {
      printf("Error %d:SetSecurityDescriptorDacl",GetLastError());
      LocalFree((HLOCAL) pNewACL);
      return(FALSE);
   }

   // STEP 16: Set the SD to the File

   if (!SetFileSecurity(pFileName, DACL_SECURITY_INFORMATION,psdNewSD))
   {
      printf("Error %d:SetFileSecurity\n",GetLastError());
      LocalFree((HLOCAL) pNewACL);
      return(FALSE);
   }

   // STEP 17: Free the memory allocated for the new ACL

   LocalFree((HLOCAL) pNewACL);
   return(TRUE);
}
```

NOTE: Security descriptors have two possible formats: self-relative and absolute. GetFileSecurity() returns an SD in self-relative format, but SetFileSecurity() expects and absolute SD. This is one reason that the code must create a new SD and copy the information, instead of simply modifying the SD from GetFileSecurity() and passing it to SetFileSecurity(). It is possible to call MakeAbsoluteSD() to do the conversion, but there may not be enough room in the current ACL anyway, as mentioned above.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

How to Add an SNMP Extension Agent to the NT Registry

PSS ID Number: Q128729

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SUMMARY

=======

After developing a new extension agent DLL, you must configure the registry so that the SNMP extension agent is loaded when the SNMP service is started. This article shows you how.

MORE INFORMATION

================

You can use REGEDT32.EXE to configure the registry, or you can have your SNMP extension agent installation program configure the registry using the Win32 registry APIs.

To configure an SNMP extension agent in the registry, follow these steps:

1. Walk down:

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
   CurrentControlSet\
   Services\
   SNMP\
   Parameter\
   ExtensionAgents
```

   You'll notice at least one entry like this:

```
1:REG_SZ:SOFTWARE\Microsoft\LANManagerMIB2Agent\CurrentVersion
```

   Add an entry for the new extension agent. For the SNMP Toaster sample in the SDK, the entry is:

```
3:REG_SZ:SOFTWARE\CompanyName\toaster\CurrentVersion
```

   This entry provides a pointer to another registery entry (see step 2) that contains the physical path where the extension agent DLL can be found. Note that "CompanyName" and "toaster" strings can be any other meaningful strings that will be used in Step 2.

2. Go to:

```
HKEY_LOCAL_MACHINE\SOFTWARE
```

   Create keys that correspond to the new entry in step 1:

CompanyName\toaster\CurrentVersion

3. Assign the path of the extension agent DLL as the value to the CurrentVersion key in step 2. For the SNMP toaster sample agent DLL, the entry is:

   Pathname:REG_SZ:D:\mstools\samples\snmp\testdll\testdl.dll

4. Note that names and values in the NT registry are case sensitive.

5. Restart the SNMP service from the control panel. The new extension agent DLL will be loaded. Event Viewer in the administrative tools can be used to view errors encountered during the startup process of the SNMP service and extension agents.

REFERENCES
==========

SNMP.TXT in the \BIN directory of the Win32 SDK.

Windows NT Resource Guide, Chapters 10-14.

Additional reference words: 3.50
KBCategory: kbnetwork
KBSubcategory: NtwkSnmp

How to Add Windows 95 Controls to Visual C++ 2.0 Dialog Editor

PSS ID Number: Q125686

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
- Microsoft Visual C++ version 2.0

SUMMARY

When using Visual C++ version 2.0 on Windows 95, the Windows 95 common controls do not appear by default on the control palette in the Visual C++ Dialog Editor. Many of these controls may be added, however, by adding an entry into the registry.

MORE INFORMATION

To add buttons in the Dialog Editor's control palette for TreeView, ListView, HotKey, Trackbar, Progress, and UpDown controls:

1. Run REGEDIT from the start menu.

2. Select:

   HKEY_CURRENT_USER\Software\Microsoft\Visual C++ 2.0 Dialog Editor

3. Select the New, Binary Value option from the Edit Menu.

4. Rename the new entry "ChicagoControls" without the quotation marks.

5. Select Modify from the edit menu to change the value of ChicagoControls to 01 00 00 00. The editor will add the spaces between each pair of digits.

6. Exit REGEDIT.

7. Restart Visual C++ version 2.0.

Additional reference words: 2.00 4.00
KBCategory: kbui
KBSubcategory: UsrCtl

How to Address Multiple CDAudio Devices in Windows NT

PSS ID Number: Q137579

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.5, 5.51, 4.0

SUMMARY
=======

To use more than one CDAudio device in Windows 3.1, you had to change the System.ini file. For more information on this process, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q82469
   TITLE   : Using Multiple CD-ROM Drives on One Machine

Now, in Windows NT 3.5 and Windows 95, support for multiple CDAudio devices has been added to the MCI CDAudio driver. However, the problem now becomes one of how to address a particular CDAudio device.

MORE INFORMATION
================

MCI solves the problem of how to control a particular CDAudio device. All you need to do is open the CDAudio device, specifying the drive letter of the CD-ROM drive to be used as the element to open. The MCI string syntax to do this is as follows:

   open d: type cdaudio alias cd

In this case, the CD-ROM drive is drive D:

The following code fragment demonstrates the equivalent MCI command syntax code:

```
    MCI_OPEN_PARMS  mciOpen;
    TCHAR           szElementName[4];
    TCHAR           szAliasName[32];
    DWORD           dwFlags;
    DWORD           dwAliasCount = GetCurrentTime();
    DWORD           dwRet;
    TCHAR           chDrive;

    chDrive = TEXT("D"); // Use drive D

    ZeroMemory( &mciOpen, sizeof(mciOpen) );
    mciOpen.lpstrDeviceType = (LPTSTR)MCI_DEVTYPE_CD_AUDIO;
    wsprintf( szElementName, TEXT("%c:"), chDrive );
    wsprintf( szAliasName, TEXT("CD%lu:"), dwAliasCount );

    mciOpen.lpstrElementName = szElementName;
```

```
    mciOpen.lpstrAlias = szAliasName;

    dwFlags = MCI_OPEN_ELEMENT | MCI_OPEN_SHAREABLE | MCI_OPEN_ALIAS |
              MCI_OPEN_TYPE | MCI_OPEN_TYPE_ID | MCI_WAIT;

    dwRet = mciSendCommand(0, MCI_OPEN, dwFlags, (DWORD)(LPVOID)&mciOpen);

    if ( dwRet == MMSYSERR_NOERROR ) {

        // The device was opened successfully

    }
    else {

        // The device was not opened successfully

    }
```

Additional reference words: 3.50 4.00 Windows 95
KBCategory: kbmm kbprg kbcode
KBSubcategory: MMCDROM

How to Allow OS/2 Programs to Run Across Logons

PSS ID Number: Q137861

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51

SUMMARY

=======

When you launch an OS/2 application from a service in Windows NT, the OS/2 application terminates when the current user logs off. If you want the application to continue running after the user logs off, you can:

- Run the OS/2 application from the Windows NT Resource Kit utility called SRVANY.

  -or-

- Launch Os2.exe directly from the service, and pass the /S switch. The syntax of the call would be:

  ```
  CreateProcess (
     NULL,
     "OS2.EXE /S /P <full path to exe> /C <command line>",
     . . .
     );
  ```

  The <command line> includes the name of the executable. For example:

  ```
     CreateProcess (
        NULL,
        "OS2.EXE /S /P C:\\OS2BINS\\LS.EXE /C LS *.c",
        . . .
        );
  ```

In either case, you must make sure that the first OS/2 application started after booting your system is the one you want to run as a service.

MORE INFORMATION

================

The OS/2 Subsystem automatically shuts down when a user logs off. Using the /S switch on the Os2.exe command line tells the OS/2 Subsystem that it is running in the context of a service. The Subsystem will then continue to run across logons.

The reason the application must be the first OS/2 executable launched is that if an OS/2 Subsystem server is present, it will be used rather than starting a new one to run as a service.

Any OS/2 applications launched by the interactive user will still be terminated upon logoff.

Additional reference words: 3.50 os2 os2ss service
KBCategory: kbprg kbhowto
KBSubcategory: BseService

How to Assign Privileges to Accounts for API Calls

PSS ID Number: Q131144

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51

SUMMARY

Some new security API calls were added to Win32 in Windows NT version 3.51. Two of these new calls, LogonUser() and CreateProcessAsUser(), require that the calling process have certain privileges. If the calling process is a service running in the Local System account, it will already have these privileges. Otherwise, the required privileges can be added to an account by using the "User Rights Policy" dialog box. Run the User Manager and choose User Rights from the Policies menu to see the dialog box.

NOTE: You must select the "Show Advanced User Rights" check box to see the privileges mentioned in this article.

MORE INFORMATION

The Win32 API reference documents the required privileges, but it gives their internal string names instead of the display names. The "User Rights Policy" dialog box displays the privileges using the display names.

The following table shows the display names associated with the internal string names:

Privilege	Display Name
SeTcbPrivilege	Act as part of the operating system
SeAssignPrimary	Replace a process level token
SeIncreaseQuota	Increase quotas

Additional reference words:
KBCategory: kbprg
KBSubcategory: BseSecurity

How to Avoid Palette Flash During Playback of Video Overlays

PSS ID Number: Q135367

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, 4.0
- Microsoft Video for Windows Development Kit (VfWDK) version 1.1

SUMMARY

Developers of multimedia titles that overlay background bitmaps with AVI (audio-visual interleaved) files are frequently surprised by what is often called "palette flash." The flash or palette re-realization can be eliminated with careful design and consideration during authoring.

MORE INFORMATION

The Windows Palette Manager attempts to satisfy the demands of the foreground window first (the AVI), and leaves background windows to contend for any unused palette entries. Windows that use identical palette entries do not flash, so the key is to author the background image and the AVI with the same palette. Incorporate the playback palette from the AVI into the background image. This works because most video Codecs (compressors and decompressors) can suggest optimal playback palettes.

Additional reference words: 4.00 1.10 3.50 dib
KBCategory: kbmm kbprg
KBSubcategory: MMVideo

How to Back Up the Windows NT Registry

PSS ID Number: Q128731

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51

SUMMARY

=======

This article shows by example how to back up portions of the Windows NT registry to be restored later.

MORE INFORMATION

================

This is normally accomplished by enabling the SeBackupPrivilege and calling RegSaveKey. The operation can fail with ERROR_ACCESS_DENIED if the caller does not have access to portions of the key, such as the registry key HKEY_LOCAL_MACHINE\SECURITY.

If you do not have access to the key, but have backup privilege, pass the REG_OPTION_BACKUP_RESTORE flag to RegCreateKeyEx in the dwOptions parameter. This has an effect similar to FILE_FLAG_BACKUP_SEMANTICS with CreateFile, allowing you to open the key for backup. The resultant key handle can be used in a subsequent call to RegSaveKey.

To back up the registry from the root, it is necessary to enumerate the subkeys from the root, opening each subkey with RegCreateKeyEx and saving the subkey with RegSaveKey.

Sample Code

The following sample source code saves the HKEY_LOCAL_MACHINE registry key, with each subkey saved to a filename matching the subkey name.

The folloing function performs the save operation:

```
LONG SaveRegistrySubKey(
    HKEY hKey,              // handle of key to save
    LPTSTR szSubKey,        // pointer to subkey name to save
    LPTSTR szSaveFileName   // pointer to save path/filename
    )
```

If the function succeeds, the return value is ERROR_SUCCESS.
If the function fails, the return value is an error value.

```
/* Save HKEY_LOCAL_MACHINE registry key, each subkey saved to a file of
 * name subkey
 *
```

```
 * this allows us to get around security restrictions which prevent
 * the use of RegSaveKey() on the root key
 *
 * the use of REG_OPTION_BACKUP_RESTORE is not documented in the Win32
 * documentation at this time.  The documentation will be changed to
 * reflect this flag.  This flag is contained in the WINNT.H header file.
 *
 * the optional target machine name is specified in argv[1]
 *
 * v1.21
 * Scott Field (sfield) 01-Apr-1995
 */

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

LONG SaveRegistrySubKey(HKEY hKey, LPTSTR szSubKey, LPTSTR szSaveFileName);
void PERR(LPTSTR szAPI, DWORD dwLastError);

int main(int argc, char *argv[])
{
   TOKEN_PRIVILEGES tp;
   HANDLE hToken;
   LUID luid;
   LONG rc;     // contains error value returned by Regxxx()
   HKEY hKey;   // handle to key we are interested in
   LPTSTR MachineName=NULL; // pointer to machine name
   DWORD dwSubKeyIndex=0;   // index into key
   char szSubKey[_MAX_FNAME]; // this should be dynamic.
                              // _MAX_FNAME is good because this
                              // is what we happen to save the
                              // subkey as
   DWORD dwSubKeyLength=_MAX_FNAME; // length of SubKey buffer

/*
    if (argc != 2) // usage
    {
        fprintf(stderr,"Usage: %s [<MachineName>]\n", argv[0]);
        return RTN_USAGE;
    }
 */

    // set MachineName == argv[1], if appropriate
    if (argc == 2) MachineName=argv[1];

    //
    // enable backup privilege
    //
    if(!OpenProcessToken(GetCurrentProcess(),
                         TOKEN_ADJUST_PRIVILEGES,
```

```
                    &hToken ))
    {
        PERR("OpenProcessToken", GetLastError() );
        return RTN_ERROR;
    }

    if(!LookupPrivilegeValue(MachineName, SE_BACKUP_NAME, &luid))

    {
        PERR("LookupPrivilegeValue", GetLastError() );
        return RTN_ERROR;
    }

    tp.PrivilegeCount           = 1;
    tp.Privileges[0].Luid       = luid;
    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(TOKEN_PRIVILEGES),
                              NULL, NULL );

    if (GetLastError() != ERROR_SUCCESS)
    {
        PERR("AdjustTokenPrivileges", GetLastError() );
        return RTN_ERROR;
    }

    // only connect if a machine name specified
    if (MachineName != NULL)
    {
        if((rc=RegConnectRegistry(MachineName,
                                  HKEY_LOCAL_MACHINE,
                                  &hKey)) != ERROR_SUCCESS)
        {
            PERR("RegConnectRegistry", rc);
            return RTN_ERROR;
        }
    }
    else hKey=HKEY_LOCAL_MACHINE;

    while((rc=RegEnumKeyEx(
                        hKey,
                        dwSubKeyIndex,
                        szSubKey,
                        &dwSubKeyLength,
                        NULL,
                        NULL,
                        NULL,
                        NULL)
                        ) != ERROR_NO_MORE_ITEMS) { // are we done?

    if(rc == ERROR_SUCCESS)
   {
        LONG lRetVal; // return value from SaveRegistrySubKey

#ifdef DEBUG
```

```
        fprintf(stdout,"Saving %s\n", szSubKey);
#endif

        // save registry subkey szSubKey to filename szSubKey
        if( (lRetVal=SaveRegistrySubKey(hKey, szSubKey, szSubKey)
            ) != ERROR_SUCCESS)
        {
            PERR("SaveRegistrySubKey", lRetVal);
        }

        // increment index into the key
       dwSubKeyIndex++;

       // reset buffer size
        dwSubKeyLength=_MAX_FNAME;

        // Continue the festivities
        continue;
    }
    else
   {
       //
       // note: we need to watch for ERROR_MORE_DATA
       // this indicates we need a bigger szSubKey buffer
       //
        PERR("RegEnumKeyEx", rc);
        return RTN_ERROR;
   }

    } // RegEnumKeyEx

    // close registry key we have been working with
    RegCloseKey(hKey);

    // Revoke all privileges this process holds (including backup)
    AdjustTokenPrivileges( hToken, TRUE, NULL, 0, NULL, NULL);

    // close handle to process token
    CloseHandle(hToken);

    return RTN_OK;
}

LONG SaveRegistrySubKey(
    HKEY hKey,              // handle of key to save
    LPTSTR szSubKey,        // pointer to subkey name to save
    LPTSTR szSaveFileName   // pointer to save path/filename
    )
{
    HKEY hKeyToSave;    // Handle of subkey to save
    LONG rc;            // result code from RegXxx
    DWORD dwDisposition;

    if((rc=RegCreateKeyEx(hKey,
                          szSubKey, // Name of subkey to open
```

```
                        0,
                        NULL,
                        REG_OPTION_BACKUP_RESTORE, // in winnt.h
                        KEY_QUERY_VALUE, // minimal access
                        NULL,
                        &hKeyToSave,
                        &dwDisposition)
                        ) == ERROR_SUCCESS)
    {
        // Save registry subkey.  If the registry is remote, files will
        // be saved on the remote machine
        rc=RegSaveKey(hKeyToSave, szSaveFileName, NULL);

        // close registry key we just tried to save
        RegCloseKey(hKeyToSave);
    }

    // return the last registry result code
    return rc;
}

void PERR(
    LPTSTR szAPI,       // pointer to failed API name
    DWORD dwLastError   // last error value associated with API
    )
{
    LPTSTR MessageBuffer;
    DWORD dwBufferLength;

    //
    // TODO get this fprintf out of here!
    //
    fprintf(stderr,"%s error! (rc=%lu)\n", szAPI, dwLastError);

    if(dwBufferLength=FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                                    FORMAT_MESSAGE_FROM_SYSTEM,
                                    NULL,
                                    dwLastError,
                                    LANG_NEUTRAL,
                                    (LPTSTR) &MessageBuffer,
                                    0,
                                    NULL))
    {


        DWORD dwBytesWritten;

        //
        // Output message string on stderr
        //
        WriteFile(GetStdHandle(STD_ERROR_HANDLE),
                  MessageBuffer,
                  dwBufferLength,
                  &dwBytesWritten,
                  NULL);
```

```
        //
        // free the buffer allocated by the system
        //
        LocalFree(MessageBuffer);
    }
}
```

Additional reference words: 3.50
KBCategory: kbprg kbcode
KBSubcategory: BseMisc BseSecurity CodeSam

How to Broadcast Messages Using NetMessageBufferSend()

PSS ID Number: Q131458

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51

The NetMessageBufferSend() API can be used to broadcast a message. To broadcast a copy a of a particular message to all workstations running messanger service in a particular domain, the LPWSTR msgname parameter needs to be specified as "DOMAINNAME*" - where DOMAINNAME is the name of the domain to which a message is to be sent. In this case, you can use the following piece of code to call this API:

```
#define UNICODE
#define MESGLEN 50
WCHAR awcToName[] = TEXT("DomainName*");
WCHAR awcFromName[] = Text("MyComputer");
WCHAR awcMesgBuffer[MESGLEN] = Text("This ia Test Message");
NET_API_STATUS nasStatus;

nasStatus = NetMessageBufferSend(NULL,
                                 awcToName,
                                 awcFromName,
                                 awcMesgBuffer,
                                 MESGLEN);
```

Additional reference words: 3.50 3.51 LanMan
KBCategory: kbnetwork kbnetwork
KBSubcategory: NtwkMisc

How to Build a Japanese NT 3.50 Application on US NT 3.50

PSS ID Number: Q126744

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5 and 3.51
- Microsoft Visual C++, 32-bit Edition, version 2.0
- Microsoft Windows NT versions 3.5 and 3.51

Use the following steps to build a Japanese application on U.S. Windows NT version 3.5 using the English 32-bit Edition of Visual C++ version 2.0:

1. Copy c_932.nls from your Japanese Windows NT version 3.5 CD or disks.

2. Add c_932.nls to your English Windows NT version 3.5 system directory (for example, WINNT35\SYSTEM32.

3. Using REGEDT32.EXE, search for "codepage" in the following registry subkey:

   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet

   Add this value:

   932 : REG_SZ : c_932.nls

4. Add the "/C932" switch to the RC compiler option in the Project setting in the Microsoft Visual C++ IDE (Intergrated Development Environment).

Take similar steps if you want to use U.S. Windows NT version 3.5 as the development environment and build an application that requires a codepage that is not present on U.S. Windows NT version 3.5.

Additional reference words: 3.50 3.51 msvcj setlocale language
KBCategory: kbprg
KBSubcategory: WIntlDev

How to Build Localized Win32s Kits with .nls Files

PSS ID Number: Q149972

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with Microsoft Win32s version 1.30c

---

SUMMARY

Microsoft Win32s version 1.30c contains files that allow localized Win32s kits to be built.

MORE INFORMATION

There are 111 files under the NLS directory in Win32s 1.30c. These include various .nls files, and the following files in nine different languages:

```
COMCTL32.DLL
COMDLG32.DLL
FTSRCH.DLL
OLEDLG.DLL
RICHED32.DLL
W32S.386
W32SYS.DLL
WINHLP32.EXE
```

These nine different languages are:

```
Danish     (extension: DAN)
German     (extension: DEU)
Spanish    (extension: ESP)
Finnish    (extension: FIN)
French     (extension: FRA)
Italian    (extension: ITA)
Dutch      (extension: NLD)
Norwegian  (extension: NOR)
Swedish    (extension: SVE)
```

NOTE: The language version of the file is signified by the extension of the file name.

To build a localized Win32s, install English Win32s as usual, and then substitute:

```
COMCTL32.DLL
COMDLG32.DLL
FTSRCH.DLL
OLEDLG.DLL
RICHED32.DLL
W32S.386
```

W32SYS.DLL
WINHLP32.EXE

with the corresponding file in the appropriate language. Then copy the NLS files in the NLS directory.

For example, to build a Danish Win32s, install Win32s English version, and then substitute:

COMCTL32.DLL
COMDLG32.DLL
FTSRCH.DLL
OLEDLG.DLL
RICHED32.DLL
W32S.386
W32SYS.DLL
WINHLP32.EXE

with

COMCTL32.DAN
COMDLG32.DAN
FTSRCH.DAN
OLEDLG.DAN
RICHED32.DAN
W32S.DAN
W32SYS.DAN
WINHLP32.DAN

Change the extension into DLL or EXE or 386 as appropriate.

Additional reference words: 1.30c Win32s NLS
KBCategory: kbprg kbhowto
KBSubcategory: wintldev

How to Calculate Dialog Base Units with Non-System-Based Font

PSS ID Number: Q125681

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

This article shows how to calculate the dialog base unit in Windows for the dialog box using a font other than System Font. You can use this calculation to build dialog templates in memory or calculate dialog dimensions.

MORE INFORMATION

Each dialog box template contains measurements that specify the position, width, and height of the dialog box and the controls it contains. These measurements are device independent, so an application can use a single template to create the same dialog box for all types of display devices. This ensures that a dialog box will have the same proportions and appearance on all screens despite differing resolutions and aspect ratios between screens.

Further, dialog box measurements are given in dialog base units. One horizontal base unit is equal to one-fourth of the average character width for the system font. One vertical base unit is equal to one-eighth of the average character height for the system font. An application can retrieve the number of pixels per base unit for the current display by using the GetDialogBaseUnits function. The low-order word of the return value, from the GetDialogBaseUnits function, contains the horizontal base units and the high-order word of the return value, from the GetDialogBaseUnits function, contains the vertical base units.

Using this information, you can compute the dialog base units for a dialog using font other than system font:

```
horz pixels == (horz dialog units * average char width  of font) / 4
vert pixels == (vert dialog units * average char height of font) / 8
```

As the font of a dialog changes, the actual size and position of a control also changes.

- Four pixels is half the average character width of the system font.
- Eight pixels is half the average character height of the system font.

Here's another version of the same formulas:

```
horz pixels == 2 * horz dialog units * (average char width  of dialog font
```

```
                        / average char width of system font)
vert pixels == 2 * vert dialog units * (average char height of dialog font
                        / average char height of system font)
```

One dialog base unit is equivalent to the number of pixels per dialog unit which gives:

```
1 horz dialog base unit == (2 * average char width  dialog font /
                              average char width  system font) pixels
1 vert dialog base unit == (2 * average char height dialog font /
                              average char height system font) pixels
```

Average character width and height of a font can be computed as follows:

```
hFontOld = SelectObject(hdc,hFont);
GetTextMetrics(hdc,&tm);
GetTextExtentPoint32(hdc,"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrst"
            "uvwxyz",52,&size);
avgWidth = (size.cx/26+1)/2;
avgHeight = (WORD)tm.tmHeight;
```

The tmAveCharWidth field of the TEXTMETRIC structure only approximates the actual average character width (usually it gives the width of the letter 'x') and so the true average character width must be calculated to match the value used by the system.

Additional reference words: 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlg

How to Calculate the Height of Edit Control to Resize It

PSS ID Number: Q124315

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK), version 3.1
- Microsoft Win32 Software Development Kit (SDK), versions 3.5 and 3.51

SUMMARY
=======

When a program changes the font of an edit control, it must calculate the new height of the control so that text is displayed correctly. When an edit control contains a border, the control automatically adds white space around the text so the text won't interfere with the border.

This article shows by example how a program can modify the height of an edit control so that text displayed in the control looks right after the program changes the font.

MORE INFORMATION
================

The height of the control on creation is calculated as the height of the control's font plus half of the smaller of the height of the control's font or the height of the system font. You can use a function similar to the one below to calculate the new height of an edit control when the font in the control is changed.

Sample Code

```
void ResizeEdit(HWND hwndEdit, HFONT hNewFont)
{
   HFONT       hSysFont,
               hOldFont;
   HDC         hdc;
   TEXTMETRIC  tmNew,
               tmSys;
   RECT        rc;
   int         nTemp;

   //get the DC for the edit control
   hdc = GetDC(hwndEdit);

   //get the metrics for the system font
   hSysFont = GetStockObject(SYSTEM_FONT);
   hOldFont = SelectObject(hdc, hSysFont);
   GetTextMetrics(hdc, &tmSys);

   //get the metrics for the new font
   SelectObject(hdc, hNewFont);
   GetTextMetrics(hdc, &tmNew);
```

```
   //select the original font back into the DC and release the DC
   SelectObject(hdc, hOldFont);
   DeleteObject(hSysFont);
   ReleaseDC(hwndEdit, hdc);

   //calculate the new height for the edit control
   nTemp = tmNew.tmHeight + (min(tmNew.tmHeight, tmSys.tmHeight)/2) +
   (GetSystemMetrics(SM_CYEDGE) * 2);

   //re-size the edit control
   GetWindowRect(hwndEdit, &rc);
   MapWindowPoints(HWND_DESKTOP, GetParent(hwndEdit), (LPPOINT)&rc, 2);
   MoveWindow( hwndEdit,
               rc.left,
               rc.top,
               rc.right - rc.left,
               nTemp,
               TRUE);
}
```

Additional reference words: 3.10 3.50 win16sdk
KBCategory: kbprg kbcode
KBSubcategory: UsrCtl

How To Cause Inheriting of Environment Variables on Windows 95

PSS ID Number: Q152648

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 3.5

SUMMARY
=======

When launching 16-bit MS-DOS applications or batch files on Windows 95 from a 32-bit process, a changed or modified environment will not be inherited if creation of the MS-DOS application requires a new console window. To cause the environment to be successfully inherited, the Win32 process can launch a 32-bit console application which will inherit the Win32 parent's current or specified environment variables. The 32-bit console application can then launch the 16-bit MS-DOS application or batch file, causing the MS-DOS application to inherit the Win32 parent's console window and its environment variables.

MORE INFORMATION
================

The CreateProcess API's lpEnvironment parameter allows programmers to specify an environment string for the process to be launched. If no environment is specified, the child process will inherit the parent's current environment variables.

On Windows 95, the parent's modified environment or specified environment string will not be inherited if the child is a 16-bit MS-DOS application or batch file. If the parent process is a Win32 console application, the 16-bit MS-DOS application or batch file can inherit the Win32 process' console and successfully inherit the environment variables of the parent.

For console-less Win32 applications to launch MS-DOS applications or batch files with an inherited or specified environment string, they must first launch a Win32 console process that will successfully inherit the environment. The Win32 console application can then launch the MS-DOS application or batch file, causing the spawned 16-bit application to inherit the Win32 parent's console and its environment variables.

Additional reference words: 3.50
KBCategory: kbprg kbhowto
KBSubcategory: Console

How to Change Hard Error Popup Handling in Windows NT

PSS ID Number: Q128642

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51

SUMMARY

In an unattended environment, you may want to automatically dispatch hard error popups that require user intervention. This article gives you the code you need to change the hard error popup mode.

MORE INFORMATION

Windows NT allows the user to change the handling of hard error popups that result from application and system errors. Such errors include no disk in the drive and general protection (GP) faults.

Normally, these events cause a hard error popup to be displayed, which requires user intervention to dispatch. This behavior can be modified so that such errors are logged to the Windows NT event log. When the error is logged to the event log, no user intervention is necessary, and the system provides a default handler for the hard error. The user can examine the event log to determine the cause of the hard error.

Registry Entry

The following registry entry controls the hard error popup handling in Windows NT:

```
HKEY_LOCAL_MACHINE\
 SYSTEM\
 CurrentControlSet\
 Control\
 Windows\
 ErrorMode
```

Valid Modes

The following are valid values for ErrorMode:

- Mode 0

  This is the default operating mode that serializes the errors and waits for a response.

- Mode 1

  If the error does not come from the system, this is the normal operating mode. If the error comes from the system, this logs the error to the event log and returns OK to the hard error. No intervention is required and the popup is not seen.

- Mode 2

  This always logs the error to the event log and returns OK to the hard error. Popups are not seen.

In all modes, system-originated hard errors are logged to the system log. To run an unattended server, use mode 2.

Sample Code to Change Hard Error Popup Mode

The following function changes the hard error popup mode. If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE.

```
BOOL SetGlobalErrorMode(
    DWORD dwErrorMode   // specifies new ErrorMode value
    )
{
    HKEY hKey;
    LONG lRetCode;

    // make sure the value passed isn't out-of-bounds
    if (dwErrorMode > 2) return FALSE;

    if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,
                    "SYSTEM\\CurrentControlSet\\Control\\Windows",
                    0,
                    KEY_SET_VALUE,
                    &hKey) != ERROR_SUCCESS) return FALSE;

    lRetCode=RegSetValueEx(hKey,
                           "ErrorMode",
                           0,
                           REG_DWORD,
                           (CONST BYTE *) &dwErrorMode,
                           sizeof(DWORD) );

    RegCloseKey(hKey);

    if (lRetCode != ERROR_SUCCESS) return FALSE;

    return TRUE;
}
```

Sample Code to Obtain Hard Error Popup Mode

The following function obtains the hard error popup mode. If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. If the function succeeds, dwErrorMode contains the error popup mode. Otherwise, dwErrorMode is undefined.

```
BOOL GetGlobalErrorMode(
    LPDWORD dwErrorMode // Pointer to a DWORD to place popup mode
    )
{
    HKEY hKey;
    LONG lRetCode;
    DWORD cbData=sizeof(DWORD);

    if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,
                    "SYSTEM\\CurrentControlSet\\Control\\Windows",
                    0,
                    KEY_QUERY_VALUE,
                    &hKey) != ERROR_SUCCESS) return FALSE;

    lRetCode=RegQueryValueEx(hKey,
                             "ErrorMode",
                             0,
                             NULL,
                             (LPBYTE) dwErrorMode,
                             &cbData );

    RegCloseKey(hKey);

    if (lRetCode != ERROR_SUCCESS) return FALSE;

    return TRUE;
}
```

Additional reference words: 3.50
KBCategory: kbprg kbcode
KBSubcategory: BseErrdebug BseMisc CodeSam

How To Change Passwords Programmatically in Windows NT

PSS ID Number: Q151546

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.51, 4.00

SUMMARY
=======

This article describes how to change passwords on accounts in Windows NT programmatically. Windows NT Lan Manager API calls are used to accomplish this task.

MORE INFORMATION
================

There are several considerations that apply to changing passwords:

- Windows NT Lan Manager APIs are Unicode only. All strings passed to and returned by these functions are in Unicode form.

- When targeting a domain controller for account update operations, be sure to target the primary domain controller for the domain. The account settings are replicated by the primary domain controller to each backup domain controller as appropriate. The NetGetDCName() Lan Manager API call can be used to get the primary domain controller computer name from a domain name.

- If the caller is an administrator or account operator on the target machine/domain, the NetUserSetInfo() Lan Manager API call at info-level 1003 can be used to override the existing password. The caller does not need to know the existing password. Note that passwords can be provided for accounts during account creation time using NetUserAdd().

- If the caller is a non-administrator on the target machine/domain, the NetUserChangePassword() Lan Manager API call can be used to override the existing password. In order for this call to succeed, the caller must supply the correct current password. NetUserChangePassword() behaves differently than other Lan Manager APIs with respect to the first parameter which specifies either a domain name or machine name. If this parameter is set to NULL, the domain name of the caller is used. Keep this in mind if you intend to change passwords on accounts outside the domain of the logged-on caller. You should explicitly provide the target domain name.

Sample Code

```
/*++

Module Name:
```

```
    chngpass.c

Abstract:

   This sample changes the password for an arbitrary user on an arbitrary
   target machine.

   When targeting a domain controller for account update operations,
   be sure to target the primary domain controller for the domain.
   The account settings are replicated by the primary domain controller
   to each backup domain controller as appropriate. The NetGetDCName()
   Lan Manager API call can be used to get the primary domain controller
   computer name from a domain name.

   Username is argv[1]
   new password is argv[2]
   optional target machine (or domain name) is argv[3]
   optional old password is argv[4]. This allows non-admin password
   changes.

   Note that admin or account operator privilege is required on the
   target machine unless argv[4] is present and represents the correct
   current password.

   NetUserSetInfo() at info-level 1003 is appropriate for administrative
   override of an existing password.

   NetUserChangePassword() allows for an arbitrary user to override
   an existing password providing that the current password is confirmed.

   Link with netapi32.lib

--*/

   #include <windows.h>
   #include <stdio.h>

   #include <lm.h>

   #define RTN_OK 0
   #define RTN_USAGE 1
   #define RTN_ERROR 13

   void
   DisplayErrorText(
       DWORD dwLastError
       );

   //
   // Unicode entry point and argv
   //

   int
   __cdecl
```

```
wmain(
    int argc,
    wchar_t *argv[]
    )
{
    LPWSTR          wUserName;
    LPWSTR          wComputerName = NULL; // default to local machine
    LPWSTR          wOldPassword;
    LPWSTR          wNewPassword;
    USER_INFO_1003  pi1003;
    NET_API_STATUS  nas;

    if( argc < 3 ) {
        fprintf(stderr, "Usage: %ls <user> <new_password> "
                        "[\\\\machine | domain] [old_password]\n",
                        argv[0]);
        return RTN_USAGE;
    }

    //
    // process command line arguments
    //

    wUserName = argv[1];
    wNewPassword = argv[2];

    if( argc >= 4 && *argv[3] != L'\0' ) {

        //
        // obtain target machine name, if appropriate,
        // always in Unicode, as that is what the API takes.
        //

        if(argv[3][0] == L'\\' && argv[3][1] == L'\\') {

            //
            // target specified machine name
            //

            wComputerName = argv[3];
        }
        else {

            //
            // the user specified a domain name. Look up the PDC.
            // This is done in both password change cases to ensure the
            // same computer is targeted for the update operation.
            //

            nas = NetGetDCName(
                NULL,
                argv[3],
                (LPBYTE *)&wComputerName
                );
```

```
        if(nas != NERR_Success) {
            DisplayErrorText( nas );
            return RTN_ERROR;
        }
    }
}

if(argc == 5) {
    wOldPassword = argv[4];
} else {
    wOldPassword = NULL;
}

if(wOldPassword == NULL) {

    //
    // administrative over-ride of existing password
    //

    pi1003.usri1003_password = wNewPassword;

    nas = NetUserSetInfo(
            wComputerName,  // computer name
            wUserName,      // username
            1003,           // info level
            (LPBYTE)&pi1003,     // new info
            NULL
            );
} else {

    //
    // allows user to change their own password
    //

    nas = NetUserChangePassword(
            wComputerName,
            wUserName,
            wOldPassword,
            wNewPassword
            );
}

if(wComputerName != NULL && wComputerName != argv[3]) {

    //
    // a buffer was allocated for the PDC name. Free it.
    //

    NetApiBufferFree(wComputerName);
}

if(nas != NERR_Success) {
    DisplayErrorText( nas );
    return RTN_ERROR;
}
```

```
    return RTN_OK;
}

void
DisplayErrorText(
    DWORD dwLastError
    )
{
    HMODULE hModule = NULL; // default to system source
    LPSTR MessageBuffer;
    DWORD dwBufferLength;
    DWORD dwFormatFlags;

    dwFormatFlags = FORMAT_MESSAGE_ALLOCATE_BUFFER |
                    FORMAT_MESSAGE_IGNORE_INSERTS |
                    FORMAT_MESSAGE_FROM_SYSTEM ;

    //
    // if dwLastError is in the network range, load the message source
    //
    if(dwLastError >= NERR_BASE && dwLastError <= MAX_NERR) {
        hModule = LoadLibraryEx(
            TEXT("netmsg.dll"),
            NULL,
            LOAD_LIBRARY_AS_DATAFILE
            );

        if(hModule != NULL)
            dwFormatFlags |= FORMAT_MESSAGE_FROM_HMODULE;
    }

    //
    // call FormatMessage() to allow for message text to be acquired
    // from the system or the supplied module handle.
    //
    if(dwBufferLength = FormatMessageA(
        dwFormatFlags,
        hModule, // module to get message from (NULL == system)
        dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // default language
        (LPSTR) &MessageBuffer,
        0,
        NULL
        ))
    {
        DWORD dwBytesWritten;

        //
        // Output message string on stderr
        //
        WriteFile(
            GetStdHandle(STD_ERROR_HANDLE),
            MessageBuffer,
            dwBufferLength,
```

```
                &dwBytesWritten,
                NULL
                );

            //
            // free the buffer allocated by the system
            //
            LocalFree(MessageBuffer);
        }

        //
        // if you loaded a message source, unload it.
        //
        if(hModule != NULL)
            FreeLibrary(hModule);
    }
```

Additional reference words: 3.51 4.00 password LanMan
KBCategory: kbprg kbhowto
KBSubcategory: NtwkMisc BseSecurity CodeSam

How to Change Small Icon for FileOpen and Other Common Dialogs

PSS ID Number: Q130758

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

Applications that need to display icons on the caption bars of File Open and other Common Dialogs can do so by installing a hook function and sending the WM_SETICON message from within the hook function to the common dialog to change its small icon. Note that OFN_ENABLEHOOK flag (or relavant flags for other common dialogs) has to be set for your hook function to be called.

MORE INFORMATION

Under Windows 95, every popup or overlapped window can have two icons associated with it, a large icon used when the window is minimized and a small icon used for displaying the system menu icon.

Common Dialogs under Windows 95 do not display a small icon on their caption bars by default. If you want the application to display its own icon for the system menu, have the application install a hook funtion for that common dialog and send the WM_SETICON message when the hook callback function is called with the WM_INITDIALOG message.

The WM_SETICON message is sent to change or set the small and large icons of a window. In this case, because you are setting the small icon, wParam must be set to FALSE.

Code Sample

The following code shows how to do this for a File Open Common Dialog:

```
 // Fill in the OPENFILENAME structure to support
 // a hook and  a template (optional).

 OpenFileName.lStructSize       = sizeof(OPENFILENAME);
 OpenFileName.hwndOwner      = hWnd;
 OpenFileName.hInstance         = g_hInst;
   ...
   ...
   ...
   ...
```

```
 OpenFileName.lpfnHook          = ComDlg32HkProc;
 OpenFileName.lpTemplateName  = NULL;
 OpenFileName.Flags               = OFN_SHOWHELP |
                OFN_EXPLORER | OFN_ENABLE_HOOK;
```

Note that the lpTemplateName parameter is set to NULL. To just install a hook, one does not need a custom template. The hook function will get called if it is sepcified in the structure.

Below is the Comdlg32HkgProc hook callback funtion that chages the small icon. This code below is for the open or save as dialog boxes only.

```
BOOL CALLBACK ComDlg32HkProc(HWND hDlg,
                             UINT uMsg,
                             WPARAM wParam,
                             LPARAM lPar
{

   HWND hWndParent;
   HICON  hIcon;


   switch (uMsg)
   {
      case WM_INITDIALOG:

         hWndParent = GetParent(hDlg);

         hIcon = LoadIcon(g_hInst, "CustomIcon");

         SendMessage(hWndParent,
                     WM_SETICON,
                    (WPARAM)(BOOL)FALSE,
                    (LPARAM)(HICON)hIcon);

        return TRUE;

        break;


      default:
        break;

    }
```

NOTE: This code calls GetParent() to get the actual window handle of the common dialog box. This is done for the FileOpen and SaveAs dialog boxes only. These dialogs, when created with the OFN_EXPLORER look with a hook and a template (optional), create a seperate dialog to hold all the controls. This is the dialog handle that is passed in the hook function. The parent of this dialog is the main common dialog window, whose caption icon must be modified. The FileOpen and SaveAs dialog boxes with the old style (no OFN_EXPLORER) need not call GetParent().

All other common dialogs, such as ChooseColor and ChooseFont, behave as the

the Windows version 3.1 common dialogs behaved, so the code listed in this article does not need to call GetParent(). It can just send the WM_SETICON message to the hDlg that is passed to the hook function.

Additional reference words: 4.00 user common dialog
KBCategory: kbui kbcode
KBSubcategory: UsrCmnDlg

How to Change the International Settings Programmatically

PSS ID Number: Q126625

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The International control panel icon allows you to choose international settings for the time, date, keyboard, and so forth. You can change these settings from your application code by calling SetLocaleInfo(). Be sure to broadcast a WM_WININICHANGE message using PostMessage() so that all applications are notified of the change.

The Win32 API allows you to display the time and date in the correct international format. Use GetDateFormat() and GetTimeFormat() to get the date and time in the format specified by the user.

Additional reference words: 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrLoc WintlDev

How To Clear the z-buffer in Direct3D Retained Mode

PSS ID Number: Q152668

The information in this article applies to:

 - Microsoft DirectX 2 Software Development Kit (SDK), for Windows 95

SUMMARY
=======

The IDirect3DRMViewport::Clear() function will clear a Retained Mode viewport's z-buffer and target rendering surface. You cannot direct this function to clear the z-buffer only. If you need to clear the z-buffer without calling IDirect3DRMViewport::Clear(), then you will need to extract the immediate mode viewport from the retained mode viewport and use the low level clear functions in the immediate mode API to clear the z-buffer only.

MORE INFORMATION
================

A set of normal Retained Mode rendering instructions may look like the following:

```
   scene->Move(D3DVALUE(1.0));
   view->Clear();
   view->Render(scene);
   rmdev->Update();
```

If you want to render without clearing your target surface, you will need to remove your call to Clear(). If you do this, however, your z-buffer will not get cleared and your objects will not be rendered properly. If you obtain the immediate mode viewport associated with your retained mode viewport, you can clear the z-buffer without clearing the target. Call IDirect3DRMViewport::GetDirect3DViewport() to retrieve the Direct3D immediate mode viewport and then call the immediate mode viewport's Clear() method to clear the z-buffer. You can do this by calling IDirect3DViewport::Clear() with the D3DCLEAR_ZBUFFER flag set.

Sample Code

The following code shows how you can modify your rendering calls to achieve this functionality:

```
   LPDIRECT3DVIEWPORT d3dview;

   scene->Move(D3DVALUE(1.0));
   view->GetDirect3DViewport(&d3dview);

   int clearflags;
   D3DRECT rc;
   clearflags = D3DCLEAR_ZBUFFER;
```

```
   rc.x1 = rc.y1 = 0;
   rc.x2 = 640;
   rc.y2 = 480;
   d3dview->Clear(1, &rc, clearflags);

   view->Render(scene);
   rmdev->Update();

Additional reference words: 4.00
KBCategory: kbrpg kbhowto
KBSubcategory: Direct3D
```

How To Close Console Window When MS-DOS Application Completes

PSS ID Number: Q152682

The information in this article applies to:

- Microsoft Win32 Software Development Kit for:
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

In Windows 95, running an MS-DOS application will cause the creation of a new console window. If this application has written text to the window, such as stdout or stderr, then the console window will not close when the application completes. In order to programmatically force the console window to close, Windows 95 must be tricked into thinking that it is OK to close the console window. In order to implement this behavior, all displayed text must be removed.

MORE INFORMATION
================

In Windows 95, the default behavior of the operating system is to keep the console window of an MS-DOS application open if it has displayed text in the window. This provides end-users with an opportunity to read any messages that may have been displayed in the window before the window is closed. To close a window in this state, you must select the Close button ("X") on the Window caption or the Close Menu options from the System menu.

To programmatically close a console window for an MS-DOS application that has displayed some text to the window, you must trick Windows 95 into believing that it is OK to close the console window. You can do this by creating a .BAT file, and in this file, adding the following lines:

```
@echo off
>> DOS app
@cls
```

The first line, "@echo off," prevents .BAT file commands from being displayed in the window. After the MS-DOS application completes, the "cls" command clears away any text that may have been sent to the console window. This behavior tricks Windows 95 and enables the console window to be closed after the .BAT file has completed.

Additional reference words: 4.00 Win95
KBCategory: kbprg kbusage kbhowto
KBSubcategory: Bsecon

How to Compile CAPCPP Sample Using Visual C++ 2.x Compiler

PSS ID Number: Q137900

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, 4.0
- Microsoft Video for Windows Development Kit version 1.1
- Microsoft Visual C++, 32-bit Edition, versions 2.0, 2.1, 2.2

SUMMARY

CAPCPP is a Microsoft Foundation Class (MFC)-based sample included in the Video for Windows Development Kit. The sample was written as a 16-bit Windows-based application. However, with minor modifications, you can compile it as a 32-bit Windows-based application.

MORE INFORMATION

The CAPCPP sample comes with a project file (Capcpp.mak) created by Visual C++ version 1.5x. To convert this project for use by Visual C++ version 2.x compiler, follow these steps:

1. Use Visual C++ version 2.x to open Capcpp.mak. It will convert this project file.

2. Remove Msacm.lib and Vfw.lib from files list presented when you click Library on the Object menu.

3. Replace FAR PASCAL _export with CALLBACK in the Captevw.cpp file.

4. Remove #include "avicap.h" from the Captevw.h file.

5. Add #define _AFX_NO_BSTR_SUPPORT to the beginning of the Stdafx.h file; this removes BSTR errors. For more information about this, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q129953
   TITLE : PRB: C2371 BSTR Redefinition When VFW.H Included in MFC App

6. Add Winmm.lib, Vfw32.lib, and Msacm32.lib to the Link tab that comes up when you click Settings on the Project menu. Also, remove Vfw.lib from the Modules list.

After completing these steps, you are ready to compile the sample as a 32-bit application by using Visual C++ version 2.x.

Additional reference words: 1.10 2.00 3.50 2.1 2.20 AVICap MCIWnd video capture visualc
KBCategory: kbmm kbhowto
KBSubcategory: MMVideo

How to Compile Large Chinese or Korean Help Files

PSS ID Number: Q123331

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK version 3.5

The Extended Help Compiler is not available in Traditional/Simplified Chinese Windows version 3.1 SDK extension or in the Korean Windows version 3.1 SDK extension. Therefore, large help files in Chinese or Korean cannot be compiled under Windows version 3.1 using the extended help compiler.

However, large Chinese or Korean help files can be compiled under Windows NT, by using Chinese or Korean Help Compiler (HC31.EXE). Make sure the directory containing the appropriate HC31.EXE is in the path.

If the .HPJ file contains Compress=YES, change it to compress=MEDIUM or to compress=FALSE.

The resulting .HLP files cannot be displayed correctly under Windows NT. If you try it, you'll see random symbols in place of the Chinese or Korean characters. To examine the result of the compilation, exit from Windows NT, and start Chinese or Korean Windows. Then display the help files.

Additional reference words: fesdk hcp
KBCategory: kbprg kbtool
KBSubcategory: WIntlDev

How to Connect Local Printers to Network Print Shares

PSS ID Number: Q152551

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  - Microsoft Windows NT, version 3.51
  - Microsoft Windows 95, version 4.0

SUMMARY
=======

Prior to Windows 95 and Windows NT, network print shares were used in Windows by connecting MS-DOS devices to network resources. Often, this was accomplished by the WNetAddConnection function that was implemented by the underlying network software. This architecture limited the number of connections to the number of MS-DOS devices that could be redirected. Further, it suffered from an inherent inflexibility of print job and printer management.

The current Win32 Application Programming Interface defines a collection of functions known as the Print Spooler API. The Print Spooler API manipulates the Spooler, a component of the operating system that is always running, and manages printers and print jobs both local and remote. The Spooler and its printers are no longer tied to MS-DOS devices for access to network print shares. Any printer in Windows can now be connected (redirected) directly to network print shares by their UNC names.

MORE INFORMATION
================

Configuring a printer to be redirected to a network print share is a three-step process: The application first obtains a current PRINTER_INFO_2 structure via the GetPrinter function; the new configuration is then defined by altering the members of the PRINTER_INFO_2 structure; once modified, the structure is passed to the SetPrinter function that resets the configuration.

To configure a printer to print to a network print share, set the pPortName member of the PRINTER_INFO_2 structure to point to a valid port name string. Windows NT requires the name of the port to be in the list of ports that are returned from the EnumPorts function. On Windows 95, this string may contain the UNC path to a remote network print share or can be from this list of installed ports. Additional ports are defined by calling the AddPort function and specifying the appropriate Port Monitor.

When AddPort is used to set up a new port, a dialog provided by the Port Monitor will pop up. There is no way to prevent this dialog from being displayed. The Port Monitor uses this dialog to configure the new port with information specific to that type of port.

Although Windows NT supports the redirection of local printers to network print shares, the recommended method for printing to a network print share

on Windows NT is a Printer Connection. A Printer Connection is a direct representation of the network printer resource on the local workstation. Printing to a Connection involves Remote Procedure Calls (RPC) that offer many advantages, such as spooling on the Server and Server-supplied printer drivers. However, Printer Connections currently work only when the client and server are both Windows NT or if the network specifically supports it.

Microsoft recommends that applications running on Windows NT attempt to call AddPrinterConnection to define Network Printers to which applications would print. If AddPrinterConnection fails, applications should then attempt to use a local printer redirected to the network print share as this article describes. Use and diagnosis of AddPrinterConnection failures is beyond the scope of this article.

For more information on the use of AddPrinterConnection and how to install local printers, please see the following article in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q147202
TITLE     : How to Add a Network Printer Connection
```

Use the following steps to configure a printer with the Print Spooler functions:

1. Use OpenPrinter to obtain a handle to a printer object whose configuration can be modified by the SetPrinter function. For SetPrinter to succeed, the calling process must have administrative privileges and the printer must be opened with PRINTER_ALL_ACCESS.

2. Call GetPrinter once to determine the size of a PRINTER_INFO_2 structure for the current configuration by passing zero for the size of the buffer in the cbBuffer parameter, 2 in the Level parameter, NULL for pPrinter parameter, the handle to the opened printer in the hPrinter parameter and the address of a DWORD in the pcbNeeded parameter. GetPrinter will fail and GetLastError will return ERROR_INSUFFICIENT_BUFFER. This is an indication of success, because GetPrinter was asked to return the required buffer size by forcing a failure with too small a buffer. Under these conditions, GetPrinter returns the required buffer size in the pcbNeeded parameter. If GetLastError returns anything other than this return value, it is an indication of an actual error.

3. Allocate a buffer of size pcbNeeded and call GetPrinter to retrieve the PRINTER_INFO_2 structure for the current configuration. For this second call, the address of the allocated buffer is passed in the pPrinter parameter and the size of this buffer is passed in the cbBuffer parameter.

4. Update the members of the PRINTER_INFO_2 structure to change the printer's configuration. Generally, the pSecurityDescriptor member of the structure should be set to NULL to avoid changing the security information for the printer. String pointers such as pPortName should simply be assigned the address of the string containing the new value. SetPrinter will copy the data from this structure and any strings pointed to by members of this structure.

5. Finally, call SetPrinter, passing the handle to the open printer, 2 for the dwLevel parameter, the address of the buffer containing the modified PRINTER_INFO_2 structure, and zero in the dwCommand parameter. If SetPrinter succeeds, the contents of the PRINTER_INFO_2 structure will have been used to reconfigure the printer. If SetPrinter fails, call GetLastError to determine the cause. A typical reason for the failure of a SetPrinter call is insufficient privileges for the user to change some part of the printer's configuration.

Sample Code
==========

The following code sample demonstrates the calls to GetPrinter and SetPrinter:

```
BOOL ConnectToRemotePrinter(char *pszRemotePath, HANDLE hPrinter)
{
    PRINTER_INFO_2  *ppi = NULL;
    DWORD           dwNeeded, dwReturned;

    if (!pszRemotePath && lstrlen(pszRemotePath) == 0)
       goto Fail;

    /*
     * Manage Printer's port connections
     */
    /* Get the size required for the buffer */
    SetLastError(0);
    if (!GetPrinter(hPrinter, 2, NULL, 0, &dwNeeded))
    {
        if (GetLastError() != ERROR_INSUFFICIENT_BUFFER)
            goto Fail;
    }

    ppi = (PRINTER_INFO_2 *)malloc(dwNeeded);
    if (!ppi)
       goto Fail;

    /* Get a copy of the printer's configuration */
    if (!GetPrinter(hPrinter, 2, (LPBYTE)ppi, dwNeeded, &dwReturned))
       goto Fail;

    /* change the connection */
    ppi->pPortName = pszRemotePath;

    /* don't set the security information */
    ppi->pSecurityDescriptor = NULL;

    /* Make it so */
    if (!SetPrinter(hPrinter, 2, (LPBYTE)ppi, 0))
       goto Fail;

    /* cleanup */
    free(ppi);
```

```
        return TRUE;

    Fail:
        if (ppi)
           free(ppi);
        return FALSE;

    } /* end of function ConnectToRemotePrinter */
```

Additional reference words: 3.51 4.00 redirection remote print resource WNetCancelConnection WNetAddConnection

KBCategory: kbprint kbhowto

KBSubcategory: GdiPrn

How to Convert a Binary SID to Textual Form

PSS ID Number: Q131320

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

It may be useful to convert a binary SID (security identifier) to a readable, textual form, for display or manipulation purposes.

One example of an application that makes use of SIDs in textual form is the Windows NT Event Viewer. If the Event Viewer cannot look up the name associated with the SID of a logged event, the Event Viewer displays a textual representation of the SID.

Windows NT also makes use of textual SIDs when loading user configuration hives into the HKEY_USERS registry key.

Applications that obtain domain and user names can display the textual SID representation when the Win32 API LookupAccountSid fails to obtain domain and user information. Such a failure can occur if the network is down, or the target machine is unavailable.

Sample Code

The following sample code displays the textual representation of the SID associated with the current user. This source code converts a SID using the same algorithm that the Windows NT operating system components use.

```
/*++

A standardized shorthand notation for SIDs makes it simpler to
visualize their components:

S-R-I-S-S...

In the notation shown above,

S identifies the series of digits as an SID,
R is the revision level,
I is the identifier-authority value,
S is subauthority value(s).

An SID could be written in this notation as follows:
S-1-5-32-544

In this example,
```

```
the SID has a revision level of 1,
an identifier-authority value of 5,
first subauthority value of 32,
second subauthority value of 544.
(Note that the above Sid represents the local Administrators group)

The GetTextualSid function will convert a binary Sid to a textual
string.

The resulting string will take one of two forms.  If the
IdentifierAuthority value is not greater than 2^32, then the SID
will be in the form:

S-1-5-21-2127521184-1604012920-1887927527-19009
  ^ ^ ^^ ^^^^^^^^^^ ^^^^^^^^^^ ^^^^^^^^^^ ^^^^^
  | | |      |          |          |        |
  +-+-+------+----------+----------+--------+--- Decimal

Otherwise it will take the form:

S-1-0x206C277C6666-21-2127521184-1604012920-1887927527-19009
  ^ ^^^^^^^^^^^^^^ ^^ ^^^^^^^^^^ ^^^^^^^^^^ ^^^^^^^^^^ ^^^^^
  |       |        |      |          |          |        |
  |   Hexidecimal  |      |          |          |        |
  +----------------+------+----------+----------+--------+--- Decimal

If the function succeeds, the return value is TRUE.
If the function fails, the return value is FALSE.  To get extended
    error information, call the Win32 API GetLastError().

Scott Field (sfield)    11-Jul-95
Unicode enabled

Scott Field (sfield)    15-May-95
--*/

#define RTN_OK 0
#define RTN_ERROR 13

#include <windows.h>
#include <stdio.h>

BOOL GetTextualSid(
    PSID pSid,          // binary Sid
    LPSTR TextualSID,   // buffer for Textual representaion of Sid
    LPDWORD dwBufferLen // required/provided TextualSid buffersize
    );

int main(void)
{
    #define MY_BUFSIZE 256 // all allocations should be dynamic
    HANDLE hToken;
    TOKEN_USER ptgUser[MY_BUFSIZE];
    DWORD cbBuffer=MY_BUFSIZE;
    char szTextualSid[MY_BUFSIZE];
```

```
DWORD cbSid=MY_BUFSIZE;
BOOL bSuccess;

//
// obtain current process token
//
if(!OpenProcessToken(
            GetCurrentProcess(), // target current process
            TOKEN_QUERY,         // TOKEN_QUERY access
            &hToken              // resultant hToken
            ))
{
    fprintf(stderr, "OpenProcessToken error! (rc=%lu)\n",
        GetLastError() );
    return RTN_ERROR;
}

//
// obtain user identified by current process' access token
//
bSuccess=GetTokenInformation(
            hToken,    // identifies access token
            TokenUser, // TokenUser info type
            ptgUser,   // retrieved info buffer
            cbBuffer,  // size of buffer passed-in
            &cbBuffer  // required buffer size
            );

// close token handle.  do this even if error above
CloseHandle(hToken);

if(!bSuccess)
{
    fprintf(stderr, "GetTokenInformation error! (rc=%lu)\n",
        GetLastError() );
    return RTN_ERROR;
}

//
// obtain the textual representaion of the Sid
//
if(!GetTextualSid(
            ptgUser->User.Sid, // user binary Sid
            szTextualSid,      // buffer for TextualSid
            &cbSid             // size/required buffer
            ))
{
    fprintf(stderr, "GetTextualSid error! (rc=%lu)\n",
        GetLastError() );
    return RTN_ERROR;
}

// display the TextualSid representation
fprintf(stdout,"%s\n", szTextualSid);
```

```
    return RTN_OK;
}


BOOL GetTextualSid(
    PSID pSid,          // binary Sid
    LPTSTR TextualSid,  // buffer for Textual representaion of Sid
    LPDWORD dwBufferLen // required/provided TextualSid buffersize
    )
{
    PSID_IDENTIFIER_AUTHORITY psia;
    DWORD dwSubAuthorities;
    DWORD dwSidRev=SID_REVISION;
    DWORD dwCounter;
    DWORD dwSidSize;

    //
    // test if Sid passed in is valid
    //
    if(!IsValidSid(pSid)) return FALSE;

    // obtain SidIdentifierAuthority
    psia=GetSidIdentifierAuthority(pSid);

    // obtain sidsubauthority count
    dwSubAuthorities=*GetSidSubAuthorityCount(pSid);

    //
    // compute buffer length
    // S-SID_REVISION- + identifierauthority- + subauthorities- + NULL
    //
    dwSidSize=(15 + 12 + (12 * dwSubAuthorities) + 1) * sizeof(TCHAR);

    //
    // check provided buffer length.
    // If not large enough, indicate proper size and setlasterror
    //
    if (*dwBufferLen < dwSidSize)
    {
        *dwBufferLen = dwSidSize;
        SetLastError(ERROR_INSUFFICIENT_BUFFER);
        return FALSE;
    }

    //
    // prepare S-SID_REVISION-
    //
    dwSidSize=wsprintf(TextualSid, TEXT("S-%lu-"), dwSidRev );

    //
    // prepare SidIdentifierAuthority
    //
    if ( (psia->Value[0] != 0) || (psia->Value[1] != 0) )
    {
        dwSidSize+=wsprintf(TextualSid + lstrlen(TextualSid),
```

```
                    TEXT("0x%02hx%02hx%02hx%02hx%02hx%02hx"),
                    (USHORT)psia->Value[0],
                    (USHORT)psia->Value[1],
                    (USHORT)psia->Value[2],
                    (USHORT)psia->Value[3],
                    (USHORT)psia->Value[4],
                    (USHORT)psia->Value[5]);
    }
    else
    {
        dwSidSize+=wsprintf(TextualSid + lstrlen(TextualSid),
                    TEXT("%lu"),
                    (ULONG)(psia->Value[5]      )   +
                    (ULONG)(psia->Value[4] <<  8)   +
                    (ULONG)(psia->Value[3] << 16)   +
                    (ULONG)(psia->Value[2] << 24)   );
    }

    //
    // loop through SidSubAuthorities
    //
    for (dwCounter=0 ; dwCounter < dwSubAuthorities ; dwCounter++)
    {
        dwSidSize+=wsprintf(TextualSid + dwSidSize, TEXT("-%lu"),
                    *GetSidSubAuthority(pSid, dwCounter) );
    }

    return TRUE;
}
```

Additional reference words: Convert LookupAccountSid SID String
KBCategory: kbprg
KBSubcategory: BseSecurity BseMisc CodeSam

How to Convert a File Path to an ITEMIDLIST

PSS ID Number: Q132750

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

When developing an application that interacts with the Windows 95 shell, it is often necessary to convert an arbitrary path to a file to an ITEMIDLIST. This can be accomplished using the IShellFolder::ParseDisplayName API.

MORE INFORMATION

================

Here is an example of how to use the IShellFolder interface to convert the path to the file README.TXT in the current directory to an ITEMIDLIST. The example is written in C. If the program is written using C++, accessing member functions through the lpVtbl variable is unnecessary.

```
LPITEMIDLIST  pidl;
LPSHELLFOLDER pDesktopFolder;
char          szPath[MAX_PATH];
OLECHAR       olePath[MAX_PATH];
ULONG         chEaten;
ULONG         dwAttributes;
HRESULT       hr;

//
// Get the path to the file we need to convert.
//
GetCurrentDirectory(MAX_PATH, szPath);
lstrcat(szPath, "\\readme.txt");

//
// Get a pointer to the Desktop's IShellFolder interface.
//
if (SUCCEEDED(SHGetDesktopFolder(&pDesktopFolder)))
{
    //
    // IShellFolder::ParseDisplayName requires the file name be in Unicode.
    //
    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, szPath, -1,
                        olePath, MAX_PATH);

    //
    // Convert the path to an ITEMIDLIST.
    //
    hr = pDesktopFolder->lpVtbl->ParseDisplayName(pDesktopFolder,
```

```
                                                    NULL,
                                                    NULL,
                                                    olePath,
                                                    &chEaten,
                                                    &pidl,
                                                    &dwAttributes);
    if (FAILED(hr))
    {
        // Handle error...
    }

    //
    // pidl now contains a pointer to an ITEMIDLIST for .\readme.txt. This
    // ITEMIDLIST needs to be freed using the IMalloc allocator returned
    // from SHGetMalloc().
    //

    ...
}
```

Additional reference words: 4.00 PIDL path file
KBCategory: kbui
KBSubcategory: UsrShell

How to Create 3D Controls in Client Area of Non-Dialog Window

PSS ID Number: Q130763

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

An application that uses standard Windows controls (edit boxes, list boxes, and so on) as part of a dialog box with the DS_3DLOOK style set, have the 3D look by default. But if an application creates child controls as part of the main window's client area, these controls do not have the 3D look by default. Applications should add the WS_EX_CLIENTEDGE style to the list of styles while creating the child controls to get the new 3D look.

MORE INFORMATION

WS_EX_CLIENTEDGE is the new extended style that gives controls (or any window for that matter) the new 3D look. When controls are created as part of a dialog box by using a dialog template based in the resource file, Windows adds the WS_EX_CLIENTEDGE style to the list of styles.

Some applications use controls as child windows in the client area of the main window. If the WS_EX_CLIENTEDGE style is not specified, these controls have the Windows version 3.11 user interface (2D look).

Use CreateWindowEx() to create controls in the client area of a non-dialog window, and make sure you OR in the WS_EX_CLIENTEDGE style. If your application is using Microsoft Foundation Classes (MFC) version 3.x, you can override the CreateEx() member function of the CEdit, CList, or any other standard control class.

Additional reference words: 4.00 95 user
KBCategory: kbui
KBSubcategory: UsrCtl

How to Create a Topmost Window

PSS ID Number: Q81137

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Windows 3.1 introduces the concept of a topmost window that stays above all the non-topmost windows even when the window is not the active window.

There are two ways to add the topmost attribute to a window:

1. Use CreateWindowEx to create a new window. Specify WS_EX_TOPMOST as the value for the dwExStyle parameter.

2. Call SetWindowPos, specifying an existing non-topmost window and HWND_TOPMOST as the value for the hwndInsertAfter parameter.

SetWindowPos can also be used to remove the topmost attribute from a window. To do so, specify HWND_NOTOPMOST or HWND_BOTTOM as the value for the hwndInsertAfter parameter.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

How To Create Hard Symbolic Links in Windows NT

PSS ID Number: Q153181

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 3.51, 4.0

SUMMARY

Windows NT supports hard symbolic links on NTFS disk volumes for Posix compatibility. This article describes how to use the two programmatic approaches that exist for creating hard symbolic links in Windows NT:

1. Write a Posix application that uses the link() call. Use the Posix development files included with the Win32SDK to build a Posix application.

2. Use the Win32 BackupWrite() API call to construct a hard symbolic link.

Hard symbolic links are only supported on files that span the same NTFS volume. A link cannot be created that spans volumes.

MORE INFORMATION

Sample Code

```
   /*++

   Copyright (c) 1996  Microsoft Corporation

   Module Name:

      ln.c

   Abstract:

   This module illustrates how to use the Win32 BackupWrite() API to
   create hard symbolic links.

   NOTE: The new link filename path must be supplied to the BackupWrite()
   Win32 API call in Unicode.

   --*/

   #ifndef UNICODE
   #define UNICODE
   #define _UNICODE
   #endif

   #include <windows.h>
```

```
#include <stdio.h>

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

int
__cdecl
wmain(
    int argc,
    wchar_t *argv[]
    )
{
    LPTSTR FileSource;
    WCHAR FileLink[ MAX_PATH + 1 ];
    LPWSTR FilePart;

    HANDLE hFileSource;

    WIN32_STREAM_ID StreamId;
    DWORD dwBytesWritten;
    LPVOID lpContext;
    DWORD cbPathLen;
    DWORD StreamHeaderSize;

    BOOL bSuccess;

    if(argc != 3) {
        printf("Usage: %ls <existing_source_file> <link_file>\n",
            argv[0]);
        return RTN_USAGE;
    }

    FileSource = argv[1];

    //
    // open existing file that we link to
    //

    hFileSource = CreateFile(
        FileSource,
        FILE_WRITE_ATTRIBUTES,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        NULL, // sa
        OPEN_EXISTING,
        0,
        NULL
        );

    if(hFileSource == INVALID_HANDLE_VALUE) {
        printf("CreateFile (source) error! (rc=%lu)\n", GetLastError());
        return RTN_ERROR;
    }

    //
```

```
    // validate and sanitize supplied link path and use the result
    // the full path MUST be Unicode for BackupWrite
    //

    cbPathLen = GetFullPathNameW( argv[2], MAX_PATH, FileLink, &FilePart
);

    if(cbPathLen == 0) {
        printf("GetFullPathName error! (rc=%lu)\n", GetLastError());
        return RTN_ERROR;
    }

    cbPathLen = (cbPathLen + 1) * sizeof(WCHAR); // adjust for byte
count

    //
    // it might also be a good idea to verify the existence of the link,
    // (and possibly bail), as the file specified in FileLink will be
    // overwritten if it already exists
    //

    //
    // prepare and write the WIN32_STREAM_ID out
    //

    lpContext = NULL;

    StreamId.dwStreamId = BACKUP_LINK;
    StreamId.dwStreamAttributes = 0;
    StreamId.dwStreamNameSize = 0;
    StreamId.Size.HighPart = 0;
    StreamId.Size.LowPart = cbPathLen;

    //
    // compute length of variable size WIN32_STREAM_ID
    //

    StreamHeaderSize = (LPBYTE)&StreamId.cStreamName - (LPBYTE)&
        StreamId+ StreamId.dwStreamNameSize ;

    bSuccess = BackupWrite(
        hFileSource,
        (LPBYTE)&StreamId,  // buffer to write
        StreamHeaderSize,   // number of bytes to write
        &dwBytesWritten,
        FALSE,              // don't abort yet
        FALSE,              // don't process security
        &lpContext
        );

    if(bSuccess) {

        //
        // write out the buffer containing the path
        //
```

```
            bSuccess = BackupWrite(
                hFileSource,
                (LPBYTE)FileLink,   // buffer to write
                cbPathLen,          // number of bytes to write
                &dwBytesWritten,
                FALSE,              // don't abort yet
                FALSE,              // don't process security
                &lpContext
                );

            //
            // free context
            //

            BackupWrite(
                hFileSource,
                NULL,               // buffer to write
                0,                  // number of bytes to write
                &dwBytesWritten,
                TRUE,               // abort
                FALSE,              // don't process security
                &lpContext
                );
        }

        CloseHandle( hFileSource );

        if(!bSuccess) {
            printf("BackupWrite error! (rc=%lu)\n", GetLastError());
            return RTN_ERROR;
        }

        return RTN_OK;
    }
```

Additional reference words: 3.51 4.00 ln link symbolic
KBCategory: kbprg kbhowto
KBSubcategory: BseFileIo BseMisc CodeSam

How to Create Inheritable Win32 Handles in Windows 95

PSS ID Number: Q118605

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

Sometimes it is convenient for you to create an object such as a semaphore or file and then allow a child process to inherit the object's handle. This provides a means for two or more related processes to easily share an object.

Although Windows 95 does not have a security system such as the one in Microsoft Windows NT, Win32 API functions that create objects still use the SECURITY_ATTRIBUTES structure to determine whether the handle to the newly created object can be inherited. This article shows how to initialize a SECURITY_ATTRIBUTES structure to control whether an object handle can be inherited.

MORE INFORMATION

Win32 API functions that create objects require a SECURITY_ATTRIBUTES parameter to give a newly created object access-control information and to determine whether the handle to the object can be inherited.

The SECURITY_ATTRIBUTES structure contains the following members:

```
Type      Name
----      ----
DWORD     nLength;
LPVOID    lpSecurityDescriptor;
BOOL      bInheritHandle;
```

Secure Win32 operating systems such as Microsoft Windows NT use the lpSecurityDescriptor member to enforce how and by which processes an object is accessed. Because Windows 95 does not have a security system, it ignores lpSecurityDescriptor. Like Microsoft Windows NT, Windows 95 uses the bInheritHandle member to determine whether an object's handle can be inherited by child processes.

To initialize a SECURITY_ATTRIBUTES structure so that a handle can be inherited, set bInheritHandle to TRUE. The following code snippet shows how to create a mutex with an inheritable handle:

```
      // Set the length of the structure, allow the handle to be
      // inherited, and use the default security descriptor (which
      // Windows 95 will ignore, but Windows NT will use.) Then create
```

```
      // a named, initially unowned mutex whose handle can be
      // inherited.

   SECURITY_ATTRIBUTES sa;
   HANDLE              hMutex1;

   sa.nLength              = sizeof(sa);
   sa.bInheritHandle       = TRUE;
   sa.lpSecurityDescriptor = NULL;

   hMutex1 = CreateMutex(&sa, FALSE, "MUTEX1");
```

To prevent the handle from being inherited, set bInheritHandle to FALSE. The following code example demonstrates creating a mutex with a noninheritable handle:

```
      // Set the length of the structure, do not allow the handle
      // to be inherited, and use the default security descriptor
      // (which Windows 95 will ignore, but Windows NT will use).
      // Create a named, initially unowned mutex whose handle cannot
      // be inherited.

   SECURITY_ATTRIBUTES sa;
   HANDLE              hMutex1;

   sa.nLength              = sizeof(sa);
   sa.bInheritHandle       = FALSE;
   sa.lpSecurityDescriptor = NULL;

   hMutex1 = CreateMutex(&sa, FALSE, "MUTEX1");
```

You can also prevent a handle to an object from being inherited by specifying NULL in the call to Win32 object creation API function instead of specifying a pointer to a SECURITY_ATTRIBUTES structure. This is equivalent to setting bInheritHandle to FALSE and lpSecurityDescriptor to NULL. For example:

```
      // Use NULL instead of pointer to SECURITY_ATTRIBUTES
      // structure to create a named, initially unowned
      // mutex whose handle cannot be inherited. A NULL security
      // descriptor will be used by Windows NT, but ignored by
      // Windows 95.

   HANDLE hMutex1;
   hMutex1 = CreateMutex(NULL, FALSE, "MUTEX1");
```

Cross-Platform Compatibility Information

Keep in mind that while Windows 95 does not have a security system, Windows NT does. Windows 95 ignores the lpSecurityDescriptor member of the SECURITY_ATTRIBUTES, but Windows NT uses it. If access to the object needs to be controlled in a specific way on Windows NT, then the lpSecurityDescriptor should be initialized by calling the Win32 security

API functions.

Additional reference words: 4.00
KBCategory: kbprg
KBSubcategory: BseSecurity

How to Create Non-rectangular Windows

PSS ID Number: Q125669

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Windows 95 and Windows NT version 3.51 provide a new API called SetWindowRgn() that makes it easy for applications to create irregularly shaped windows.

MORE INFORMATION

================

In previous versions of Windows and Windows NT, it was possible to create only rectangular windows. To simulate a non-rectangular window required a lot of work on the application developer's part. Besides handling all drawing for the window, the application was required to perform hit-testing and force underlying windows to repaint as necessary to refresh the "transparent" portions of the window.

Windows 95 and Windows NT version 3.51 greatly simplify this by providing the SetWindowRgn function. An application can now create a region with any desired shape and use SetWindowRgn to set this as the clipping region for the window. Subsequent painting and mouse messages are limited to this region, and Windows automatically updates underlying windows that show through the non-rectangular window. The application need only paint the window as desired.

For more information on using SetWindowRgn, see the Win32 API documentation.

Additional reference words: 4.00

KBCategory: kbui

KBSubcategory: UsrMisc

How to Debug Flat Thunks

PSS ID Number: Q133722

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Debugging flat thunks generated by the thunk compiler can be difficult because the thunk mechanism is complex and debugging tools capable of tracing through thunks are difficult to use. This article presents an overall strategy for debugging flat thunks, several specific debugging techniques, and a troubleshooting guide that explains how to fix many common thunking problems.

MORE INFORMATION
================

Limitations on What a Target DLL Can Do

Before you get started debugging thunks, keep in mind that there are some limitations on what a target DLL can do inside a thunk. This is because a Win16-based application calling a Win32-based DLL is not a Win32-based process; likewise, a Win32-based application calling a Win16-based DLL is not a Win16-based process. Common specific limitations include:

- You cannot create threads inside a thunk from a Win16-based application to a Win32-based DLL.

- The code inside Win32-based DLLs called by thunks should require little stack space because the calling Win16-based processes have much smaller stacks than do Win32-based applications.

- Win16-based DLLs that contain interrupt service routines (ISRs) must not thunk to Win32-based DLLs while handling interrupts.

- Win32-based applications must not pass pointers to data located on the stack as parameters of thunks or call Win16-based DLLs that switch stacks.

Why Debugging Flat Thunks Can Be Difficult

Debugging flat thunks is difficult partly because the flat thunk mechanism is a complex part of the Windows kernel. Its complexity stems from the fact that it must transform function calls in 32-bit compiled code into calls compatible with 16-bit code and vice versa. Because 32-bit code uses different data types and CPU register sets from 16-bit code, the flat thunk

mechanism must translate function parameters, switch stacks, and translate return values. It is optimized for speed, yet must allow preemptive Win32 code to call non-preemptive Win16 code. The thunk compiler makes creating flat thunks much easier than manually creating them, but it isn't foolproof.

Debugging flat thunks is difficult not only because the mechanism itself is complex, but also because the necessary debugging tools are more difficult to master. Application-level debuggers such as the Microsoft Visual C++ debugger and WinDBG cannot trace through thunks because they consist of both 32-bit and 16-bit code and cause the system to claim or release the Win16Mutex. To trace through a thunk, you need to use a system-level debugger such as WDEB386.EXE. The major drawbacks to using WDEB386.EXE are that you need to know Intel x86 assembly language, know how Intel x86 microprocessors work, and remember many debugger commands.

The Best Strategy to Use

The best strategy for debugging thunks is to divide and conquer because is relatively easy and eliminates most of the problems before you need to trace through assembly language code in a system-level debugger. Flat thunks are composed of a Win32-based DLL and a Win16-based DLL, so it is possible to test each of these in isolation before testing them together. Create a Win16-based application to test the Win16-based DLL, and create a Win32-based application to test the Win32-based DLL. Doing so allows you to use a wide variety of debugging tools to verify that each side works properly.

Preliminary Checklist - Before Compiling with the Thunk Compiler

Once you've verified that each side works correctly, it's time to put the two together to test the thunk itself. Before you compile the thunk with the thunk compiler, make a preliminary check of the following items:

1. In your thunk script, make sure that each function has the correct number and types of parameters. Also make sure that the parameter types are supported by the thunk compiler. If they aren't, you will have to change the parameter somehow to pass the data with a supported type.

2. If you pass any structures as parameters, make sure you use the same structure packing in your Win32-based DLL, Win16-based DLL, and thunk script. You set structure packing in your C/C++ compiler's command line, and in the thunk compiler command line. Note that the thunk compiler's packing switch is lowercase for the 16-bit side, and uppercase for the 32-bit side.

3. Make sure that the functions you're thunking to are exported correctly and use the PASCAL calling convention if they're 16-bit, or _stdcall if they're 32-bit. The thunk compiler does not support the _cdecl and __fastcall calling conventions.

4. Make sure that your Win32-based DLL calls ThunkConnect32() each time its DllMain() function is called. Likewise, make sure the Win16-based DLL

has an exported DllEntryPoint() function, separate from its LibMain(), that calls ThunkConnect16() and returns TRUE if ThunkConnect16() succeeds.

NOTE: You actually call XXX_ThunkConnect16() and XXX_ThunkConnect32() where XXX is the symbol you define with the thunk compiler's -t switch. The code generated by the thunk compiler uses these symbols to generate tables that call ThunkConnect16() and ThunkConnect32.

5. Make sure that the value specified in the thunk compiler's command line -t switch is the same for both the Win32 and Win16 thunk DLLs. The value must also correspond to the prefix of the ThunkConnect calls in your Win16-based and Win32-based DLLs (see the note in step 4).

6. Verify that the Win16-based DLL has DLLEntryPoint exported with the RESIDENTNAME keyword in its module definition (.DEF) file. Without the RESIDENTNAME keyword, the ThunkConnect32/ThunkConnect16 call will fail and the DLLs will not load.

7. Verify in your Win16-based DLL's makefile that the resource compiler is marking the DLL as 4.0. If it is marked less than 4.0, it won't load and the thunk will fail.

8. If your 32-bit to 16-bit thunk function returns a pointer, make sure that the base type is the same size on both the 16-bit and 32-bit sides of the thunk. If the size of the base type is different, then the thunk compiler issues an error message stating, "Cannot return pointers to non-identical types." One way to work around this problem is to return a pointer to a different, but compatible, data type. For example, a thunk cannot return a pointer to an int because an int is two bytes on the 16-bit side, but four bytes on the 32-bit side. Change the thunk's return type from a pointer to an int to a pointer to a long in the thunk script and the source code of the Win16-based and Win32-based DLLs.

   If you write a 16-bit to 32-bit thunk that returns a pointer, the thunk compiler issues an error message stating, "Pointer types may not be returned." The thunk compiler does not allow 16-bit to 32-bit thunks to return pointer types because once the thunk has returned from the 32-bit function, the pointer will not point to data in the correct Win32-based process address space. This is because the address spaces of all Win32-based processes use the same range addresses and are preemptively context-switched.

9. If the linker reports an "unresolved external" error and the symbol is a function name that is spelled consistently throughout all source code, module definition files, and the thunk script, make sure that all occurrances of its prototype are consistent. On the Win32 side, the thunk function must be declared with the __stdcall type; on the Win16 side, the function must be declared with the PASCAL type. In C++ projects, be sure to declare and define both sides of the thunk function with the extern "C" linkage specifier in addition to the __stdcall or the PASCAL type.

Trouble-Shooting Guide - After Compiling with the Thunk Compiler

---

After you check the preliminaries, build your thunk DLLs and try to run them. If they run, continue with further testing to make sure they're rock solid. If they don't run, use the following troubleshooting guide to determine and fix the cause of the problem.

ThunkConnect16() in the Win16 or ThunkConnect32() in the Win32 side fails:

1. Run the debugging versions of the system DLLs. The debugging versions of KERNEL32.DLL and KRNL386.EXE contain many diagnostic messages to tell you why the thunk did not intialize. To run the debugging versions of the system DLLs, use the "Switch to Debug DLLs" icon in the Start Menu under Win32 SDK Tools. Use the "Switch to Non-debugging DLLs" to switch back to the retail version.

2. Verify that the Win16-based DLL has a call to ThunkConnect16() and the Win32-based DLL has a corresponding call to ThunkConnect32(). If one of these is missing, then the other will fail, and the thunk DLLs will fail to load.

3. Put breakpoints in your Win32 DLL's DllMain(), and in your Win16 DLL's DllEntryPoint() and LibMain() functions to see which DLLs are not loading.

If your ThunkConnect16() and ThunkConnect32() calls are working properly, but the thunk still isn't, it is time to simplify your thunk. You can actually attack this in two ways. First, start by removing parameters from the thunk one by one and recompiling it. Or, second, create a simple thunk that works, and build it up until it fails by following these steps:

1. Create a simple thunk and execute it just to make sure you have the thunk mechanism set up correctly. A good choice for a simple thunk is a function with no return value and no parameters. If even the simple thunk doesn't work, run through the preliminary checklist above to make sure you have things set up correctly. Then proceed with step 2.

2. Check to make sure the target DLL and any DLLs it relies on can be found and loaded. If one is missing, or the loader can't find it, the thunk won't work.

3. Make sure your target DLL isn't doing something that it can't in the context of a thunk.

Once you have a simplified thunk that works, but your real thunk still doesn't work, follow these steps:

1. Add parameters to the simple thunk one at a time to determine if a parameter is causing the failure. If one is, make sure that the parameter is the right type, that the function is declared and defined with the same number and types of parameters in both DLLs and in the thunk compiler, and that the function is declared as PASCAL or _stdcall.

2. If your target DLL is a Win16-based DLL and it can't access its global or static data, make sure you've exported the function correctly. If you

use the /GD switch with Visual C++, you must declare and define the function with the __export keyword in the Win16-based DLL's source code. Just listing the function's name in the DLL's module definition (.DEF) file is not enough because the compiler does not process the .DEF file, so it won't generate the prolog and epilog code that exported functions require.

3. If calls to LocalAlloc() in your target Win16-based DLL cause general protection (GP) faults, make sure your function is exported as described in step 2.

4. If you get a GP fault in KERNEL32 just after your target Win16-based function returns, make sure the target function is declared and defined as PASCAL. The C calling convention cannot be used. Although uncommon in C or C++ code, but more likely in assembly language, make sure that the target function didn't modify the DS, SS, BP, SI, or DI registers.

5. If you get a GP fault in your 32-bit thunk DLL or KERNEL32 immediately after your Win32-based target function returns, make sure that the target function is declared as _stdcall and that it didn't modify the DS, ES, FS, SS, EBP, EBX, ESI, or EDI registers. C or C++ code should not cause the registers to be modified, but assembly-language code should be checked carefully.

6. If your Win16-based target function returns to an invalid location, make sure it is declared and defined as FAR. This is especially important for small model DLLs; functions in medium and large model DLLs are FAR by default.

7. If you experience a GP fault in a Win16-based function when you access more than 64K of data from a pointer passed in as a parameter (that is, a thunked pointer), you need to allocate an array of tiled selectors, as described in the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q132005
   TITLE : DOCERR: AllocSelector & FreeSelector Documentation Incomplete

   On the Win16 side, thunked pointers always consist of a single selector with a limit of 64K, which means you cannot use them as huge pointers. The entire original range of data that the pointer addresses is accessible to the Win16-based target DLL - but only if it creates an array of tiled selectors to reference it, and if it uses huge pointer variables to access the data.

8. Make sure you only use a thunked pointer in the context of the thunk. Selectors allocated by the thunk compiler for use by Win16-based targets are freed as soon as the thunk returns.

9. Put breakpoints at the beginning of your target functions to make sure you're getting into them. If you are, and you've debugged the target side independently of the thunk, and the error is caused inside the target, then chances are good that the target is doing something that can't be done in a thunk, or referencing memory that doesn't exist. Please see steps 7 and 8.

Additional reference words: 4.00 95 quick
KBCategory: kbprg
KBSubcategory: BseMisc

How To Debug OLE Server Applications Using MSVC

PSS ID Number: Q151074

The information in this article applies to:

- Microsoft OLE libraries included with:
  - Microsoft Windows NT, version 3.51
  - Microsoft Windows 95, version 4.0

SUMMARY
========

An OLE client application involves interaction with other OLE server applications. This interaction could be with in-process or out-of-process servers. The client application may or may not have debugging information. These combinations make the debugging process of an OLE application complicated. This article presents some techniques that can be used for debugging OLE-enabled applications.

MORE INFORMATION
=================

Debugging an In-Process OLE Server Application

An OLE client application interacting with an in-process OLE server application is simply loading an OLE server DLL in the client application address space. To debug such an in-process server, standard DLL debugging techniques can be used. For more information, please see the following article in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q85221
TITLE     : Debugging a Dynamic-Link Library (DLL) in Windows
```

Debugging an Out-of-Process OLE Server Application

In an OLE client application interacting with an out-of-process OLE server application, the debugging involves crossing process spaces, which makes it much more difficult. Following are few techniques that can be used to debug out-of-process OLE server applications:

- Setup a hardcoded breakpoint in the server, and when the breakpoint hits in the server code, the Microsoft Visual C++ (MSVC) debugger is launched. Then step through server code and add breakpoints at locations of interest in server code.

  Add the following line of code in the source code, where a hardcoded breakpoint is needed.

  ```
  DebugBreak();
  ```

- Setup a breakpoint in the container/client application, then step into the code where the server code is called. The action of stepping into the code causes a new version of the MVSC debugger to be launched with the Server debug information. Then you can step through the server code and add breakpoints at locations of interest in the server code. To configure this, check in the MSVC option, the Tool Option, the debug tab, the Just-in-time debugging, and the OLE RPC debugging checkboxes that need to be checked. This technique requires source code and a debug version of the client application as well. For more information, please see the following article in the Microsoft Knowledge Base:

    ARTICLE-ID: Q122680
    TITLE   : Tutorial: Debugging OLE Applications

- Launch the server application from the client and then find the process id of the server application using PView. Launch the MSVC -p <process ID>. This launches the MSVC option that attaches itself to the running server application. However, this technique is not useful in debugging the startup code of the server application. For more information, please see the following article in the Microsoft Knowledge Base:

    ARTICLE-ID: Q120707
    TITLE   : How to Debug an Active Process in Visual C++ 2.0

- If the server application has registered the ClassFactory object with the system as single use (CoRegisterClassObject with CLS_SINGLEUSE flag), you can run the server application from the MSVC option as stand alone. To simulate the server being launched from the container/client application, you need to specify the /Embedding /Automation program arguments as applicable in the MSVC debug options and run the server application as stand alone. The /Embedding and /Automation switches do the appropriate server initialization, and register the class factory as if launched from the container/client application. Because the server's class factory is single use and not already connected, when the container/client application tries to hook up to the server, the container connects to the running server application in the debugger, and debugging is easier. For more information, please see the following article in the Microsoft Knowledge Base:

    ARTICLE-ID: Q117558
    TITLE   : Tutorial: Debugging OLE Client-Server Combinations

- If the server application registers the ClassFactory object with the system as multiple use (CoRegisterClassObject with CLS_MULTIPLEUSE flag), run the server application from the MSVC option as stand alone. Since such servers register the ClassFactory on startup, you do not need to specify /Embedding /Automation program arguments in the MSVC debug options. When the container/client application tries to connect to the server, it connects to the instance of the server application in the debugger because the server's class factory is registered multiple use, and debugging is easier.

Additional reference words: 3.50 3.51 4.00
KBCategory: kbole kbtshoot kbhowto
KBSubcategory: LeTwoOth

How to Decipher EMR_SELECTOBJECT Records in Enhanced Meta File

PSS ID Number: Q142319

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.5, 3.51, 4.0

SUMMARY

The EMR_SELECTOBJECT record in an enhanced metafile can indicate that the object to be selected is a stock object. It does so by setting the high order bit in the handle. This article shows how to decipher those records to get a handle to the stock object.

MORE INFORMATION

If an EMR_SELECTOBJECT record is encountered while enumerating the records in an enhanced metafile, the dParm member of the ENHMETARECORD parameter is normally an index into the handle table of the object to be selected.

However, if the high order bit of dParm[0] is set, the remaining bits indicate the index that can be used with GetStockObject() to obtain a handle to the object of interest.

For example, consider the following enumeration callback function:

```
int CALLBACK EnumEnhMetafileProc( HDC hDC,
                                  HANDLETABLE *lpHTable,
                                  ENHMETARECORD *lpEMFR,
                                  int nObj, LPARAM lpData )
{
    DWORD    dwIndex;
    HGDIOBJ  hObj;

    // Which record type is this record?
    switch( lpEMFR->iType )
    {
        case EMR_SELECTOBJECT: // It's a SelectObject() record
            // Is the high order bit set?
            if( lpEMFR->dParm[0] & 0x80000000 )
            {
                // High order bit is set - its a stock object
                // Strip the high bit to get the index
                dwIndex = lpEMFR->dParm[0] & 0x7fffffff;
                // Pass the index to GetStockObject()
                hObj = GetStockObject( dwIndex );
                // Select the object into the DC
                SelectObject( hDC, hObj );
            }
            else
```

```
            {
                // High order bit wasn't set - not a stock object
                SelectObject( hDC,
                          lpHTable->objectHandle[lpEMFR->dParm[0]] );
            }
        break;
        default:
            // Process other records
            PlayEnhMetaFileRecord( hDC, lpHTable, lpEMFR, nObj );
        break;
    }
    // Return non-zero to continue enumeration
    return 1;
}
```

Note that this information is important only if the application needs to decipher the record. Simply passing the EMR_SELECTOBJECT record to PlayEnhMetaFileRecord() is valid independent of whether or not the object of interest is a stock object.

In Windows 95, optimizations built into the operating system can result in a stock object record being recorded into the metafile even when a specific non-stock object is requested. This can happen, for example, when an application requests a solid brush of color RGB(128,128,128). The operating system recognizes this as the GRAY_BRUSH stock object and enters the stock object record into the metafile accordingly.

Additional reference words: Windows 95 4.00 3.50 Enumerate EnumEnhMetaFile EnhMetaFileProc missing EMR_CREATEBRUSHINDIRECT
KBCategory: kbgraphic kbhowto
KBSubcategory: GdiMeta

How to Delete Keys from the Windows NT Registry

PSS ID Number: Q127990

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.5

There are two ways to delete registry keys: use REGEDT32.EXE or call RegDeleteKey() from your application.

The documentation for RegDeleteKey() points out that the specified key to be deleted must not have subkeys. If the key to be deleted does have subkeys, RegDeleteKey() will fail with access denied. This happens despite the fact that the machine account has delete privileges and the registry handle passed to RegDeleteKey() was opened with delete access. The additional requirement is that the key must have no subkeys.

This limitation does not exist in 16-bit Windows. The difference exists in Windows NT because of atomicity and security considerations that 16-bit Windows does not have.

You can select a key with subkeys and delete it with REGEDT32. This is because REGEDT32 recursively deletes the subkeys for you, making multiple call to RegDeleteKey(). You should use recursive subkey deletion in your application as well, if you need to delete keys that have subkeys.

Additional reference words: 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

How to Design Multithreaded Applications to Avoid Deadlock

PSS ID Number: Q126768

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Debugging a multithreaded application that deadlocks is challenging because the debugger cannot identify for you which thread owns which resource. You would have to track this information in your code. Because it is difficult to debug a deadlock situation, it is important to design your application to avoid deadlock.

This article is a brief introduction to a very complex topic. There are references at the bottom of this article for additional information.

The key point to keep in mind when designing a multithreaded program is that resources must always be requested in the same order.

MORE INFORMATION

The Win32 API provides WaitForSingleObject() and WaitForMultipleObjects() for requesting resources with handles. You would use a different method to request other resources, depending on the resource type.

Many deadlocks occur because resources are not requested in the same order by the application threads. For example:

- Thread 1 holds resource A and wants resource B.

- Thread 2 holds resource B and wants resource A.

Both threads block forever, resulting in deadlock. There are many other possible scenarios.

To avoid this problem, identify all of your application's critical resources and order them from least precious to most precious. Design your code such that if a thread needs several resources, it requests them in order, starting with the least precious resource. Resources should be freed in the reverse order and as soon as it is possible. This is not a requirement to avoid deadlock, but it is good practice.

In the example given above, suppose that resource B is more precious than resource A. Here's how the code would resolve the situation:

- Thread 2 already holds B, but because it wants A, it releases

B and waits for A.

- Thread 1 grabs B, then begins the task. It releases A when possible.

- Thread 2 grabs A and waits for B.

- Thread 1 finishes the task, then releases B.

- Thread 2 grabs B, finishes the task, then releases A, then releases B.

The reason you should request the least precious resource first is that it doesn't matter as much if you hold it longer while waiting to acquire all the resources that you need. If the resource is precious, you want to hold it for the smallest amount of time possible, so other threads can use it.

REFERENCES
==========

MSDN Development Library, "Detecting Deadlocks in Multithreaded Win32 Applications", by Ruediger Asche.

For more information, refer to a good book on operating system design.

Additional reference words: 3.50 4.00 95 race condition
KBCategory: kbprg
KBSubcategory: BseProcThrd

How to Detect All Program Terminations

PSS ID Number: Q125689

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

The processes for detecting program terminations in fall into two categories:

- Win32 processes use WaitForSingleObject() or WaitForMultipleObjects() to wait for other Win32 processes, Windows 16-bit processes, and MS-DOS-based applications to terminate.

- Windows 16-bit processes, on the other hand, must use the TOOLHELP NotifyRegister() function. The 16-bit processes can be notified when other 16-bit processes and MS-DOS-based applications exit, but have the limitation of not being notified of Win32 process activity.

MORE INFORMATION

================

Win32 processes can use WaitForSingleObject() or WaitForMultipleObjects() to wait for a spawned process. By using CreateProcess() to launch a Win32 process, 16-bit process, or MS-DOS-based application, you can fill in a PROCESS_INFORMATION structure. The hProcess field of this structure can be used to wait until the spawned process terminates. For example, the following code spawns a process and waits for its termination:

```
STARTUPINFO StartupInfo = {0};
StartupInfo.cb = sizeof(STARTUPINFO);
if (CreateProcess(NULL, szCommandLine, NULL, NULL, FALSE,
                  0, NULL, NULL, &StartupInfo, &ProcessInfo))
{
    WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
    /* Process has terminated */
    ...
}
else
{
    /* Process could not be started */
    ...
}
```

If necessary, you can put this code into a separate thread to allow the initial thread to continue to execute.

This synchronization method is not available to 16-bit processes. Instead, they must use the TOOLHELP NotifyRegister function to register a callback function to be called when a program terminates. This method will detect the termination of 16-bit processes and MS-DOS-based applications, but not Win32 processes.

The following code shows how to register a callback function with NotifyRegister():

```
FARPROC lpfnCallback;

lpfnCallback = MakeProcInstance(NotifyRegisterCallback, ghInst);
if (!NotifyRegister(NULL, (LPFNNOTIFYCALLBACK)lpfnCallback,
                    NF_NORMAL))
{
   MessageBox(NULL, "NotifyRegister Failed", "Error", MB_OK);
   FreeProcInstance(lpfnCallback);
}
```

The next section of code demonstrates the implementation of the callback function:

```
BOOL FAR PASCAL __export NotifyRegisterCallback (WORD wID,
                                                 DWORD dwData)
{
   HTASK hTask;  // task that called the notification callback
   TASKENTRY te;

   // Check for task exiting
   switch (wID)
   {
      case NFY_EXITTASK:
         // Obtain info about the task that is terminating
         hTask = GetCurrentTask();
         te.dwSize = sizeof(TASKENTRY);
         TaskFindHandle(&te, hTask);

         // Check if the task that is terminating is our child task.
         // Also check if the hInstance of the task that is
         // terminating is the same as the hInstance of the task
         // that was WinExec'd by us earlier in the program.

         if (te.hTaskParent == ghtaskParent &&
             te.hInst == ghInstChild)
             PostMessage(ghwnd, WM_USER+509, (WORD)te.hInst, dwData);
         break;

      default:
         break;
   }
   // Pass notification to other callback functions
   return FALSE;
}
```

The NotifyRegisterCallback() API is called by the 16-bit TOOLHELP DLL in

the context of the process that is causing the event. Problems arising because of reentrancy and notification chaining makes the callback function subject to certain restrictions. For example, operations that cause TOOLHELP events cannot be done in the callback function. (See the TOOLHELP NotifyRegister function documentation in your Software Development Kit for events that cause TOOLHELP callbacks.)

There is no way a 16-bit process can be notified when a Win32 process exits. However, a 16-bit process can use TaskFirst() and TaskNext() to periodically walk the task list to determine if a Win32 process is still executing. This technique also works for 16-bit processes and MS-DOS-based applications. For example, the following code shows how to check for the existence of a process:

```
BOOL StillExecuting(HINSTANCE hAppInstance)
{
   TASKENTRY  te = {0};

   te.dwSize = sizeof(te);
   if (TaskFirst(&te))
       do
       {
          if (te.hInstance == hAppInstance)
             return TRUE;      // process found
       } while (TaskNext(&te));

   // process not found
   return FALSE;
}
```

Refer to the TermWait sample for complete details on how to use NotifyRegister and implement a callback function. For additional information, please search in the Microsoft Knowledge Base using this word:

```
TERMWAIT
```

Additional reference words: 4.00 95 end exit notification notify spawn terminate termination
KBCategory: kbprg kbcode
KBSubcategory: BseProcThrd

How to Detect If a Color Is a Dithered Color

PSS ID Number: Q139201

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s versions 1.2, 1.25a, and 1.3

SUMMARY
=======

When you use a COLORREF (color) to draw into a display context (DC), sometimes you need to ensure that the color is a solid color rather than a dithered color.

MORE INFORMATION
================

If you need to test to see if a particular COLORREF is a solid color, you can compare the COLORREF against the return value of GetNearestColor() to see if they are equivalent.

For example, the following function returns TRUE if the COLORREF value that is passed in exists as a solid color in the specified display context:

```
BOOL IsSolidColor(HDC hDC, COLORREF crColor)
{
    if (crColor == GetNearestColor(hDC, crColor)) {

      // Color is solid
      return TRUE;

    } else {

      // Color is dithered
      return FALSE;

    }
}
```

In some operations such as creating a brush, you may want to force GDI to use a solid color. To be sure that you get a solid color, you can use the GetNearestColor() function to retrieve the solid color that best matches a specified logical color. Then use that color to create the solid brush.

For example, CreateSolidBrush(crRGB) will not guarantee a solid brush. However, CreateSolidBrush(GetNearestColor(hDC, crRGB)) will guarantee a

solid brush for the device for which you passed the DC.

Additional reference words: 1.20 1.30 3.10 4.00 3.50 dither colorref
KBCategory: kbprg kbhowto
KBSubcategory: W32

How to Detect If a Screen Saver Is Running on Windows NT

PSS ID Number: Q150785

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

SUMMARY
=======

In Windows NT, it is useful to determine whether a screen saver is running on the system. Currently, there is no direct way to detect this state.

However, there is an indirect approach to determine if a screen saver is running by testing for the existence of a desktop named "screen-saver" that the winlogon component of Windows NT creates for the screen saver to execute on. This desktop is created and destroyed when the screen saver is started and stopped.

When you use the OpenDesktop() Win32 API to attempt to open the desktop named "screen-saver", three scenarios result that help to determine whether a screen saver is running:

1. The OpenDesktop() call succeeds: Close the returned desktop handle with the CloseDesktop() Win32 API. A screen saver is running.

2. The OpenDesktop() call fails and GetLastError() indicates ERROR_ACCESS_DENIED: The screen-saver desktop exists, but the caller does not have access to the desktop. A screen saver is running.

3. The OpenDesktop() call fails, and GetLastError() does not indicate ERROR_ACCESS_DENIED: If the error is ERROR_FILE_NOT_FOUND, the desktop does not exist and a screen saver is not running. Otherwise, an error occurred attempting to open the desktop. A screen saver is not running.

WORKAROUND
==========

Sample Code

```
/**
    The following sample illustrates how to detect if a screen saver
    is running on Windows NT. This sample works by checking for the
    existence of a desktop named "screen-saver". This desktop is
    created dynamically by winlogon when a screen saver needs to be
    launched.
**/

#include <windows.h>
#include <stdio.h>

BOOL IsScreenSaverRunning( void );
```

```
int
__cdecl
main(
    void
    )
{
    if(IsScreenSaverRunning()) {
        printf("Screen saver is running!\n");
    }
    else {
        printf("Screen saver is NOT running!\n");
    }

    return 0;
}

//
// returns TRUE if a screen saver is running, or FALSE if not.
//

BOOL
IsScreenSaverRunning(
    void
    )
{
    HDESK hDesktop;

    //
    // try to open the desktop that the screen saver runs on. This
    // desktop is created on the fly by winlogon, so it only exists
    // when a screen saver is invoked.
    //

    hDesktop = OpenDesktop(
        TEXT("screen-saver"),   // desktop name where screen saver runs
        0,
        FALSE,
        MAXIMUM_ALLOWED         // open for all possible access
        );

    if(hDesktop == NULL) {

        //
        // if the call fails due to access denied, the screen saver
        // is running because the specified desktop exists - we just
        // don't have any access.
        //

        if(GetLastError() == ERROR_ACCESS_DENIED) return TRUE;

        //
        // otherwise, indicate the screen saver isn't running
        //
```

```
            return FALSE;
        }

        //
        // successfully opened the desktop (the screen saver is running)
        //

        CloseDesktop(hDesktop);

        return TRUE;
    }
```

Additional reference words: 3.50 3.51
KBCategory: kbprg kbhowto
KBSubcategory: GdiScrsav BseMisc CodeSam

How to Detect Slow CPU & Unaccelerated Video Under Windows 95

PSS ID Number: Q131259

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

Under Windows 95, use GetSystemMetrics(SM_SLOWMACHINE) to check for low-end computers. It returns a nonzero value if the computer has a 386 CPU, is low on memory, or has a slow display card.

The return values (bit flags) are:

0x0001	- CPU is a 386
0x0002	- low memory machine (less than 5 megabytes)

The following is notable for video:

0x0004	- slow (nonaccerated) display card

Additional reference words: 4.00 win95 system display slow machine
KBCategory: kbui
KBSubcategory: UsrSys

How to Detect the Number of Colors Available in a DC

PSS ID Number: Q139202

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s versions 1.2, 1.25a, and 1.3

SUMMARY

When writing an application that uses the GDI, you may sometimes need to detect how many colors are available in the current video mode.

MORE INFORMATION

If you need to detect the total number of bits that are used to form a pixel in a given display context, you can use the following algorithm:

```
int dBitsInAPixel = GetDeviceCaps(hdc, PLANES) *
                    GetDeviceCaps(hdc, BITSPIXEL);
```

To determine how many colors are available, you can raise 2 to the power of dBitsInAPixel and this will return the maximum number of colors that are displayable at a given time in the specified display context (DC).

Additional reference words: 1.20 1.30 3.10 4.00 3.50 colorref rgb bpp
KBCategory: kbprg kbhowto
KBSubcategory: W32

How to Determine If a Device Is Palette Capable

PSS ID Number: Q72387

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

On a device that supports the Windows palette management function, an application can create a logical palette, select the palette into a device context (DC), and realize the palette, which maps its colors into the system (hardware) palette. The GetDeviceCaps() API informs an application whether a given device is capable of performing palette manipulation and, if so, the size of the palette. This article discusses the GetDeviceCaps() API and how it is used.

MORE INFORMATION

To determine whether a device can perform palette operations, call the GetDeviceCaps() API with the RASTERCAPS parameter. If the RC_PALETTE bit of the return is set, the device supports the palette management functions.

To determine how many colors in the system palette are available for the application to use, the following parameters can be used in a GetDeviceCaps() call:

Parameter	Description
SIZEPALETTE	Total number of entries in the system palette
NUMRESERVED	Number of reserved (static) colors in the system palette

This functionality is demonstrated in the MyPal sample code that is included on the Windows version 3.x Software Development Kit (SDK) Source Code 2 disk. For a demonstration, start MyPal and click the right mouse button.

The reserved colors are entries in the system palette that are used by Windows and cannot be changed by an application [except by using SetSystemPaletteUse(), which is not recommended]. The reserved colors are used for the following purposes:

- Active border

Active caption
Background color MDI applications
Desktop background color
Push button faces
Push button edges
Push button text
Caption text
Disabled (gray) text
Highlight color in controls (to highlight items in the control)
Highlight text color
Inactive border
Inactive caption
Inactive caption text (new to Windows version 3.1)
Menu background
Menu text
Scroll-bar gray area
Window background
Window frame
Window text

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiPal

How to Determine the Japanese OEM Windows Version

PSS ID Number: Q130054

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Win32s version 1.2

SUMMARY

Because of the variety of computer manufacturers (NEC, Fujitsu, IBMJ, and so on) in Japan, sometimes Windows-based applications need to know which OEM (original equipment manufacturer) manufactured the computer that is running the application. This article explains how.

MORE INFORMATION

There is no documented way to detect the manufacturer of the computer that is currently running an application. However, a Windows-based application can detect the type of OEM Windows by using the return value of the GetKeyboardType() function.

If an application uses the GetKeyboardType API, it can get OEM ID by specifying "1" (keyboard subtype) as argument of the function. Each OEM ID is listed here:

OEM Windows	OEM ID
Microsoft	00H (DOS/V)
all AX	01H
EPSON	04H
Fujitsu	05H
IBMJ	07H
Matsushita	0AH
NEC	0DH
Toshiba	12H

Application programs can use these OEM IDs to distinguish the type of OEM Windows. Note, however, that this method is not documented, so Microsoft may not support it in the future version of Windows.

As a rule, application developers should write hardware-independent code, especially when making Windows-based applications. If they need to make a hardware-dependent application, they must prepare the separated program file for each different hardware architecture.

Additional reference words: 3.10 1.20 3.50 1.20 kbinf
KBCategory: kbhw
KBSubcategory: wintldev

How to Determine the Type of Handle Retrieved from OpenPrinter

PSS ID Number: Q126258

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

OpenPrinter returns a valid handle when a printer name or a server name is passed to it. Sometimes it may be necessary to determine if the returned handle is a handle to a printer because some Win32 spooler functions only accept printer handles and will fail on server handles. The following code determines if a handle is a printer handle:

```
BOOL IsPrinterHandle( HANDLE hPrinter)
{
    DWORD       cbNeeded;
    DWORD       Error;
    BOOL        bRet = FALSE;
    LPPRINTER_INFO_2 pPrinter;
    DWORD       cbBuf;
    HANDLE      hMem = NULL;

    if( !GetPrinter(hPrinter, 2, (LPBYTE)NULL, cbBuf, &cbNeeded ))
    {
        Error = GetLastError( );

        if( Error == ERROR_INSUFFICIENT_BUFFER )
        {
            hMem = GlobalAlloc(GHND, cbNeeded);
            if (!hMem) return bRet;
            pPrinter = (LPPRINTER_INFO_2)GlobalLock(hMem);
            cbBuf = cbNeeded;
            if(GetPrinter(hPrinter, 2, (LPBYTE)pPrinter, cbBuf, &cbNeeded))
            {
                bRet = TRUE;
                GlobalUnlock(hMem);
                GlobalFree(hMem);
            }
            else SetLastError( ERROR_INVALID_PRINTER_NAME );
        }
        else if( Error == ERROR_INVALID_HANDLE )
        {
            SetLastError( ERROR_INVALID_PRINTER_NAME );
        }
    }
    return bRet;
}
```

Additional reference words: 3.10 3.50 4.00 95

KBCategory: kbprint kbcode

KBSubcategory: GdiPrn

How to Determine Which Version of Win32s Is Installed

PSS ID Number: Q121091

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, and 1.2

SUMMARY

The "Win32s Programmer's Reference" describes how to use the GetWin32sInfo() function in a program to determine which version of Win32s is installed. This article explains how to determine which version of Win32s is installed interactively.

MORE INFORMATION

Use one of the following two methods to determine interactively which version of Win32s is installed:

1. Check the WIN32S.INI file in your Windows system directory. The Version entry contains the major version and the build number (m.mm.bbb.b). This entry should be modified by any Win32-based application which installs a later version of Win32s on your Windows machine.

   NOTE: Because it is up to the application vendor to set this value when installing Win32s, the value may not be accurate. Microsoft strongly urges all independent software vendors (ISVs) to modify the WIN32S.INI file so that this information is available to customers.

   -or-

2. If Win32s is installed on top of Windows for Workgroups, select the WIN32S16.DLL file from the Windows system directory in File Manager. Then from the File menu, choose Properties. The Version line contains the major version and the build number.

Version Information

Win32s Release versions and corresponding build numbers:

Win32s Release #	Build #
1.1	1.1.88
1.1a	1.1.89
1.15	1.15.103
1.15a	1.15.111
1.2	1.2.123
1.25	1.2.141
1.25a	1.2.142
1.30	1.2.159

1.30a	1.2.166

OLE versions shipped with corresponding Win32s versions:

Release #	OLE32	OLE16
1.2	2.02	2.02
1.25	2.03	2.03
1.25a	2.03a	2.03a
1.30	2.03b	2.03a
1.30a	2.03c	2.03b

Additional reference words: 1.10 1.15 1.20 1.25 1.25a 1.30 1.30a
KBCategory: kbenv
KBSubCategory: W32s

How To Disable Initial RAS Wizard for Phonebook Extensions

PSS ID Number: Q151095

The information in this article applies to:

- Microsoft Remote Access Service for Windows 95

SUMMARY
=======

By using Remote Access Service (RAS) phonebook extensions, it is possible to programmatically create phonebook entries for Windows 95's dial-up networking applet. However, these APIs do not allow you to disable the automatic Windows 95 phonebook Entry Wizard. If you do not want Windows 95 to invoke the Connection Wizard to create the first phonebook entry, your application must modify the registry.

MORE INFORMATION
================

On a fresh installation of Windows 95 with Dial-Up Networking installed, the RAS phonebook has no entries. When a user opens the Dial-Up Networking applet for the first time, a Wizard appears, instructing the user to create the first phonebook entry "My Connection." The Wizard appears even if another application already created the first phonebook entry.

To disable the initial Connection Wizard on Windows 95 manually, you can run REGEDIT.EXE and set HKEY_CURRENT_USER\RemoteAccess\Wizard to the following series of bytes: 80 00 00 00.

The code below shows how to disable the Dial-Up Networking Connection Wizard programmatically. However, these registry modifications only apply to Windows 95. Your application must determine the operating system you are running on. All other versions of Windows and Windows NT, both current and future, will not support this registry entry. For more information on version checking, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q92395
   TITLE   : Determining System Version from a Win32-based Application

Sample Code

```
   #include <windows.h>
   #include <stdio.h>

   void main (void)
       {
       LONG rc;
       HKEY hKey;
       DWORD dwVal;
```

```
        printf ("Setting HKEY_CURRENT_USER\\RemoteAccess\\Wizard to"
                " 0x00000080.\n This will stop the wizard from appearing "
               "when the dial-up\nnetworking applet is opened.\n");

        //
        // Open the registry key
        //
        rc = RegOpenKeyEx (HKEY_CURRENT_USER, "RemoteAccess",
                           0, KEY_SET_VALUE, &hKey);

        if (rc != ERROR_SUCCESS)
            {
            printf ("RegOpenKeyEx failed with return code %i\n", rc);
            return;
            }

        //
        // Modify the key
        //
        dwVal = 0x00000080;
        rc = RegSetValueEx (hKey, "wizard", 0, REG_BINARY,
                            (LPVOID) &dwVal, sizeof (dwVal));

        if (rc != ERROR_SUCCESS)
            printf ("RegSetValueEx failed with return code %i\n", rc);

        //
        // Cleanup
        //
        RegCloseKey (hKey);
        }
```

Additional reference words: 4.00 win95 ras wizard registry remote access
KBCategory: kbprg kbhowto
KBSubcategory: NtwkRAS

How to Disable the Screen Saver Programmatically

PSS ID Number: Q126627

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51

SUMMARY

=======

Under Windows NT, you can disable the screen saver from your application code. To detect if the screen saver is enabled, use this:

```
SystemParametersInfo( SPI_GETSCREENSAVEACTIVE,
                      0,
                      pvParam,
                      0
                    );
```

On return, the parameter pvParam will point to TRUE if the screen saver setting is enabled in the System control panel applet and FALSE if the screen saver setting is not enabled.

To disable the screen saver setting, call SystemParametersInfo() with this:

```
SystemParametersInfo( SPI_SETSCREENSAVEACTIVE,
                      FALSE,
                      0,
                      SPIF_SENDWININICHANGE
                    );
```

MORE INFORMATION

================

When the screen saver is activated by the system, it is run on a desktop other than the user's desktop (similar to the login desktop displayed when no one is logged in). Therefore, you cannot use FindWindow() to determine if the screen saver is currently active.

Here are two methods that you can use to detect if the screen saver is currently running:

1. Get the name of the current screen saver from the registry, parse the PE header of the screen saver binary to get the process name, then check for an active process with that name in the performance registry.

   -or-

2. Write a screen saver that would be spawned by the system and would in turn spawn the "real" screen saver. The first screen saver could notify your application when the screen saver has been activated or

deactivated.

Additional reference words: 3.50
KBCategory: kbgraphic
KBSubcategory: GdiScrsav

How to Display Debugging Messages in Windows 95

PSS ID Number: Q125868

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SUMMARY

In Windows 95, 16-bit and 32-bit Windows-based applications may use OutputDebugString() to display debug messages. Furthermore, the 16-bit and 32-bit system DLLs may also display debug messages. This article describes how to view these messages during application developement.

MORE INFORMATION

It is possible to use the 16-bit DBWIN application to display debug messages from 16-bit Windows-based applications and from the debugging versions of system DLLs (such as GDI.EXE, USER.EXE, and KRNL386.EXE). To receive debug messages via DBWIN, you must install the Windows 95 SDK debug components.

To receive messages from 32-bit Windows-based applications under Windows 95, you must debug the application with a Win32 debugger such as WinDbg, or install WDEB386 as a .VxD or in the AUTOEXEC.BAT file. To receive messages from the debugging versions of the 32-bit system DLLs (KERNEL32.DLL, USER32.DLL, and GDI32.DLL), you must install the Windows 95 SDK debugging components, in conjunction with WDEB386.

You can use WDEB386 to display debug messages from both 16-bit and 32-bit Windows-based applications and from the debugging versions of system components. Because WDEB386 works over a serial communications port, it is necessary to use a serial terminal or second computer to operate it. For more information about configuring and using WDEB386, please search for articles in the Microsoft Knowledge Base by using this word:

   WDEB386

Alternative system level debuggers, which provide functionality similiar to WDEB386, may in the future be provided by third-party vendors.

Also, you can write 32-bit application-level debuggers that display debug messages from the debuggee by handling the DEBUG_EVENT structure member OUTPUT_DEBUG_STRING_EVENT.

Additional reference words: 3.95 4.00
KBCategory: kbprg
KBSubcategory: BseErrdebug

How to Display Old-Style FileOpen Common Dialog in Windows 95

PSS ID Number: Q131282

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY
=======

If you want an application to revert back to the old-style FileOpen or SaveAs common dialog box, you must either provide a dialog template or a hook function in addition to not specifying the OFN_EXPLORER flag.

MORE INFORMATION
==========

Windows 95 provides a new flag for the File Open or Save As common dialog box called OFN_EXPLORER. When set, this flag ensures that the File Open dialog box displays a user interface that is similar to the Windows Explorer (or so-called Explorer-style dialog box).

You may want your application to revert to the old Windows version 3.1 style dialog box. For example, you might want to maintain a user interface consistent with the Windows NT user interface. Windows NT version 3.51 currently does not support the new Explorer-style File Open common dialog box. The next version of Windows NT, however, should implement this new feature.)

To display the old-style common dialog box:

- Don't specify the OFN_EXPLORER value in the OPENFILENAME structure's Flags member.

- Provide a dialog template or a hook. If the application does not have either one, a simple hook that always returns FALSE should suffice.

NOTE: When you specify a hook function for the old-style common dialog box in Windows 95, the hDlg received in the hookProc is the actual handle to the dialog containing the standard controls. This is not true, however, for the new Explorer-style common dialog hookProc where the hDlg received is the handle to a child of the dialog box containing the standard controls. Therefore, to get a handle to the actual File Open or Save As dialog box, an application should call GetParent() on the hDlg passed to the hook procedure.

Additional reference words: 4.00 subdialog
KBCategory: kbui
KBSubcategory: UsrCmnDlg

How to Draw a Custom Window Caption

PSS ID Number: Q99046

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Microsoft Windows draws captions in the caption bar (or title bar) for all eligible windows in the system. Applications need to specify only the WS_CAPTION style to take advantage of this facility. The current version of Microsoft Windows, however, imposes three significant restrictions on the captions. An application that does not want to be tied by any of these restrictions may want to draw its own caption. This article lists the restrictions and the steps required to draw a window caption.

These restrictions also apply to Windows NT, but there are a few differences for Windows 95.

It is important to note that an application should not draw its own caption unless it has very good reasons to do so. A window caption is a user interface object, and rendering it in ways different from other windows in the system may obstruct the user's conceptual grasp of the Microsoft Windows user interface.

MORE INFORMATION

Windows and Windows NT

The three important restrictions imposed by Microsoft Windows version 3.1 and Microsoft Windows NT on the caption for a window are:

- It consists of text only; graphics are not allowed.
- All text is centered and drawn with the system font.
- The length of the displayed caption is limited to 78 characters even when there is space on the caption bar to accommodate extra characters.

An application can essentially render its own caption consisting of any graphic and text with the standard graphics and text primitives by painting on the nonclient area of the window. The application should draw in response to the WM_NCPAINT, WM_NCACTIVATE, WM_SETTEXT, and WM_SYSCOMMAND messages. When processing these messages, an application should first pass on the message to DefWindowProc() for default processing, then render its

caption before returning from the message. This ensures that Microsoft Windows can properly draw the nonclient area. Because drawing the caption is part of DefWindowProc()'s nonclient area processing, an application should specify an empty window title to avoid any Windows-initiated drawing in the caption bar. The following steps indicate the computations needed to determine the caption drawing area:

1. Get the current window's rectangle using GetWindowRect(). This includes client plus nonclient areas and is in screen coordinates.

2. Get a device context (DC) to the window using GetWindowDC().

3. Compute the origin and dimensions of the caption bar. One needs to account for the window decorations (frame, border) and window bitmaps (min/max/system boxes). Remember that different window styles will result in different decorations and a different number of min/max/system boxes. Use GetSystemMetrics() to compute the dimensions of the frame, border, and the system bitmap dimensions.

4. Render the caption within the boundaries of the rectangle computed in step 3. Remember that the user can change the caption bar color any time by using the Control Panel. Some components of the caption, particularly text backgrounds, may need to be changed based on the current caption bar color. Use GetSysColor to determine the current color.

The following code sample draws a left-justified caption for a window (the code sample applies only to the case where the window is active):

Sample Code

```
   case WM_NCACTIVATE:
      if ((BOOL)wParam == FALSE)
      {
         DefWindowProc( hWnd, message, wParam, lParam );
      // Add code here to draw caption when window is inactive.

         return TRUE;
      }
      // Fall through if wParam == TRUE, i.e., window is active.

      case WM_NCPAINT:
      // Let Windows do what it usually does. Let the window caption
      // be empty to avoid any Windows-initiated caption bar drawing

         DefWindowProc( hWnd, message, wParam, lParam );
         hDC = GetWindowDC( hWnd );
         GetWindowRect( hWnd, (LPRECT)&rc2 );

      // Compute the caption bar's origin. This window has a system box
      // a minimize box, a maximize box, and has a resizeable frame

         x = GetSystemMetrics( SM_CXSIZE ) +
             GetSystemMetrics( SM_CXBORDER ) +
```

```
            GetSystemMetrics( SM_CXFRAME );
         y = GetSystemMetrics( SM_CYFRAME );
         rc1.left = x;
         rc1.top = y;

      // 2*x gives twice the bitmap+border+frame size. Since there are
      // only two bitmaps, two borders, and one frame at the end of the
      // caption bar, subtract a frame to account for this.

         rc1.right = rc2.right - rc2.left - 2*x -
                     GetSystemMetrics( SM_CXFRAME );
         rc1.bottom = GetSystemMetrics( SM_CYSIZE );

      // Render the caption. Use the active caption color as the text
      // background.

         SetBkColor( hDC, GetSysColor(COLOR_ACTIVECAPTION) );
         DrawText( hDC, (LPSTR)"Left Justified Caption", -1,
                 (LPRECT)&rc1, DT_LEFT );
       ReleaseDC( hWnd, hDC );
       break;
```

Windows 95

On Windows 95, the text is not centered and the user can choose the Font. In addition, your application might want to monitor the WM_WININICHANGED message, because the user can change titlebar widths, and so forth, dynamically. When this happens, the application should take the new system metrics into account, and force a window redraw.

Additional reference words: 3.00 3.10 3.50 4.00 minimum maximum
KbCategory: kbui kbcode
KbSubCategory: UsrPnt

How to Draw a Gradient Background

PSS ID Number: Q128637

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

This article provides source code for drawing a gradient background pattern similar to the one used in Microsoft Setup applications. The code will compile and run on Windows version 3.1, Win32s, and Windows 95.

MORE INFORMATION
================

WARNING: ANY USE BY YOU OF THE CODE PROVIDED IN THIS ARTICLE IS AT YOUR OWN RISK. Microsoft provides this code "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and/or fitness for a particular purpose.

```
/************************************************************
*                                                           *
*  DrawBackgroundPattern()                                  *
*                                                           *
*  Purpose: This function draws a gradient pattern that     *
*           transitions between blue and black. This is     *
*           similar to the background used in Microsoft     *
*           setup programs.                                 *
*                                                           *
************************************************************/
void DrawBackgroundPattern(HWND hWnd)
{
  HDC hDC = GetDC(hWnd);  // Get the DC for the window
  RECT rectFill;          // Rectangle for filling band
  RECT rectClient;        // Rectangle for entire client area
  float fStep;            // How large is each band?
  HBRUSH hBrush;
  int iOnBand;  // Loop index

  // How large is the area you need to fill?
  GetClientRect(hWnd, &rectClient);

  // Determine how large each band should be in order to cover the
  // client with 256 bands (one for every color intensity level)
  fStep = (float)rectClient.bottom / 256.0f;
```

```
  // Start filling bands
  for (iOnBand = 0; iOnBand < 256; iOnBand++) {

    // Set the location of the current band
    SetRect(&rectFill,
            0,                          // Upper left X
            (int)(iOnBand * fStep),     // Upper left Y
            rectClient.right+1,         // Lower right X
            (int)((iOnBand+1) * fStep)); // Lower right Y

    // Create a brush with the appropriate color for this band
    hBrush = CreateSolidBrush(RGB(0, 0, (255 - iOnBand)));


    // Fill the rectangle
    FillRect(hDC, &rectFill, hBrush);

    // Get rid of the brush you created
    DeleteObject(hBrush);
  };

  // Give back the DC
  ReleaseDC(hWnd, hDC);
}
```

Additional reference words: 3.00 3.10 3.50 4.00 GRADIENT BACKGROUND DITHER
KBCategory: kbgraphic kbcode
KBSubcategory: GdiMisc

How to Examine the Use of Process Memory Under Win32s

PSS ID Number: Q129599

The information in this article applies to:

- Microsoft Win32s, version 1.2

SUMMARY

=======

Under Windows, tools like HeapWalk and PWalk can be used to examine memory use of 16-bit code. However, these tools cannot be used to look at memory use of 32-bit code. This article discusses how to look at process memory use under Win32s.

MORE INFORMATION

================

If you run the debug version of Win32s and kernel debugger WDEB386, you can break into the debugger at any point by pressing CTRL+C and using debug information from the Win32s VxD. Use the command .w32s to get the list of information types available.

```
#.w32s
W32S debug routines:

A - General Info
B - Print Free LS ranges
C - Print RRD & Section lists
D - Print Modules list
E - Toggle SwapOut trace
F - Toggle PageFault trace
G - count present alias pages
H - List RRD Commit List
I - Toggle Virtual Alloc/Free trace
J - Toggle Mapped Section trace
K - List Locked Pages
[ESC] Exit W32S Debug Routines
```

Option C gives you information about the sparse memory usage.

The memory for .EXE and .DLL files is allocated in the sparse memory. Here's an example printout using option C:

```
RRD List:
Index      Start      Size       Owner      #Commits   CommSize   #PresPg
00000000   87AA0000   0000E000   00000000   - VIEW -   - VIEW -   - VIEW -
00000001   87A90000   00001000   00001F37   00000001   00001000   00000001
00000002   87A50000   00040000   00000000   00000000   00000000   00000000
00000003   87A40000   00002000   00000000   - VIEW -   - VIEW -   - VIEW -
00000004   87A30000   00009000   00000000   - VIEW -   - VIEW -   - VIEW -
00000005   87A20000   00002000   00000000   - VIEW -   - VIEW -   - VIEW -
00000006   87A10000   00002000   00000000   - VIEW -   - VIEW -   - VIEW -
```

```
00000007   87910000   00100000   00001F37   00000001   00001000   00000001
00000008   878F0000   00020000   00001F37   00000001   00020000   00000002
00000009   878C0000   00021000   00001F37   00000001   00021000   00000001
0000000A   878B0000   00005000   00000000   00000001   00005000   00000005
0000000B   87860000   00043000   00000000   00000001   00043000   00000031
0000000C   87830000   0002D000   00000000   00000001   0002D000   0000000C
0000000D   87810000   00011000   00000000   00000001   00011000   0000000E
0000000E   80869000   00001000   00001F37   00000001   00001000   00000000
0000000F   87800000   00002000   00001F37   00000001   00002000   00000001
00000010   80635000   00001000   00001F37   00000001   00001000   00000000
                      ========                         ========   ========
Total                 00229000                         000CD000   00000056

Sections List:
SecIndex   hFile      SecSize    #Ref       #Views     CommSize   #PresPg
00000001   00000004   00002000   00000000   00000001   00002000   00000001
00000002   00000005   00002000   00000000   00000001   00002000   00000001
00000003   00000006   00009000   00000000   00000001   00009000   00000003
00000004   00000007   00002000   00000000   00000001   00002000   00000001
00000005   00000008   0000E000   00000000   00000001   0000E000   00000001
                      ========                         ========   ========
Total                 0001D000                         0001D000   00000007

G. Total                                               000EA000   0000005D
```

The Size column contains the reserved size and the CommSize column contains the committed size. The addresses are zero-based (ring 0), not based on 0xffff0000 (ring 3). Therefore, you must add 0x10000 to the addresses you see in the list in order to get the ring 3 addresses.

Option D gives you the list of modules and where they reside in memory. These addresses are zero-based addresses as well, as is any information that you get from the VxD.

Another way to get information indicating where things are placed in memory is to set the verbose loader flag (0x20) in the Win32sDebug variable in the [386Enh] section of the SYSTEM.INI file.

NOTE: Do not add the 0x, just write Win32sDebug=20. The loader then will print in the debug terminal information about each loaded module. For example:

```
Open file D:\WIN32APP\FREECELL\FREECELL.EXE in mode 0xa0
LELDR: allocating 0x11000
LELDR: Module D:\WIN32APP\FREECELL\FREECELL.EXE [1] loaded at 0x87820000
LELDR: obj  1 loaded @ 0x87821000, 0x   5c00 bytes   .text,flags=0x60000020
LELDR: obj  2 loaded @ 0x87827000, 0x      0 bytes    .bss,flags=0xc0000080
LELDR: obj  3 loaded @ 0x87828000, 0x    200 bytes  .rdata,flags=0x40000040
LELDR: obj  4 loaded @ 0x87829000, 0x    a00 bytes   .data,flags=0xc0000040
LELDR: obj  5 loaded @ 0x8782a000, 0x   2400 bytes   .rsrc,flags=0x40000040
LELDR: obj  6 loaded @ 0x8782d000, 0x    200 bytes    .CRT,flags=0xc0000040
LELDR: obj  7 loaded @ 0x8782e000, 0x    a00 bytes  .idata,flags=0x40000040
LELDR: obj  8 loaded @ 0x8782f000, 0x   1e00 bytes  .reloc,flags=0x42000040
File D:\WIN32APP\FREECELL\FREECELL.EXE is closed
```

The addresses here are ring 3 addresses.

REFERENCES
==========

Please see the "Win32s Programmer's Reference" included in the Win32 SDK for more information about the debugging features. This information is not included in the version of the Win32s documentation distributed with Visual C++.

Additional reference words: 1.20
KBCategory: kbprg
KBSubcategory: W32s

How to Extract the Profile Path from a Gina in Windows NT

PSS ID Number: Q142790

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.5, 3.51

SUMMARY
=======

This article explains how to get a profile path to return from a GINA in Windows NT.

MORE INFORMATION
================

In a standard Windows NT system, interactively logged-on users are given a profile. A profile is a registry hive that is tailored to a particular user. The profile is typically used to save user-specific information such as screen appearance, mouse click speeds, whether there is a screen saver, whether the screen saver is secure, and so on. This profile, referenced using the special registry key HKEY_CURRENT_USER, is loaded by Winlogon during the interactive boot process.

The interface between Winlogon and GINA DLLs includes information passed back from GINA that allows Winlogon to locate and load the logged-on user's profile. The WlxLoggedOutSAS() pProfile parameter is used to return a pointer to a structure of type WLX_PROFILE_V1_0.

The WLX_PROFILE_V1_0 structure is currently used to support remote and mandatory profiles, which can be configured with User Manager for Domains (usrmgr) | User properties | Profile | User Profile Path. This action can be performed programmatically through the Windows NT LAN Manager API NetUserSetInfo() at info-level 3.

If a NULL pointer is returned from WlxLoggedOutSAS() as the pProfile parameter, Winlogon will handle the loading (and creation) of the user profile. In this case, the optional, administrator-defined profile path will be ignored for the logon.

Step-by-Step Procedure

Based on the supplied username and domain name, the following steps can be used to determine if and what the administrator-defined profile path is set to:

1. Determine the current domain name, with a call to the Windows NT LAN Manager API NetUserModalsGet() at info level 2. The returned USER_MODALS_INFO_2 structure member usrmod2_domain_name will contain the domain name. This call need only be made once, during the first logon attempt. Store the result of the first call in a global variable.

The current domain name is needed for comparison against the logon domain name, in order to determine where to look up account information.

2. If the supplied domain name is not equal to the current domain name, get the computer name of the domain controller associated with the supplied domain. Use the Win32 API lstrcmpiW() for string comparison and the Windows NT LAN Manager API NetGetDCName() to obtain the computer name of the domain controller.

3. Use the Windows NT LAN Manager API NetUserGetInfo() at info-level 3 to acquire the user account information for the specified user. When calling NetUserGetInfo(), target the approriate machine -- either the local machine by way of NULL, or the domain controller computer name acquired from NetGetDCName().

4. If the USER_INFO_3 structure member usri3_profile returned by NetUserGetInfo() is an empty string, no profile was defined by an administrator. In this case, WlxLoggedOutSAS() can return a NULL pointer as the pProfile parameter, and no further actions need to be performed; otherwise continue.

5. Copy the USER_INFO_3 structure member usri3_profile to storage allocated by HeapAlloc(GetProcessHeap()...). The following source code illustrates this point:

```
HeapAlloc( GetProcessHeap(), 0,
   (lstrlenW(usri3_profile) + 1) * sizeof(WCHAR) // string + NULL
   );
```

6. Allocate a block of memory of size WLX_PROFILE_V1_0. The following source code illustrates this point:

```
HeapAlloc( GetProcessHeap(), 0, sizeof(WLX_PROFILE_V1_0) );
```

7. Set the dwType member of the WLX_PROFILE_V1_0 structure to WLX_PROFILE_TYPE_V1_0.

8. Set the pszProfile member of the WLX_PROFILE_V1_0 structure to point to the copy of the USER_INFO_3 usri3_profile structure member.

9. Return a pointer to the allocated WLX_PROFILE_V1_0 structure from WlxLoggedOutSAS() as the pProfile parameter.

NOTE: The Windows NT LAN Manager APIs are Unicode only. Strings passed to these API must be in Unicode form. Gina is also Unicode only, so this is not usually an issue.

NOTE: It is important to free buffers allocated by the Windows NT LAN Manager API. The Windows NT Lan Manager API NetApiBufferFree() can be used for this purpose.

Windows NT Version Specific Information

In Windows NT 3.51 and above, Microsoft recommends that you call the Win32

API LogonUser() to obtain an access token representing the supplied username and domain name. This API does not return information specific to any authentication package, such as the user profile path. For this reason, it is necessary to follow the steps outlined in this article.

In Windows NT 3.5, the LsaLogonUser() must be used to obtain an access token representing the supplied username and domain name. This API does return a profile path, so the steps outlined in this article do not apply. However, the interface to LsaLogonUser() is subject to change in future versions of Windows NT; LogonUser() should be used when possible.

Sample Code

```
/*
 The following function illustrates a WlxLoggedOutSAS() which obtains
 the user profile path, and returns the result if a profile path is
 found. This function is a modified version of WlxLoggedOutSAS() taken
 from the Win32 SDK version 3.51 gina sample, found in
 Mstools/Samples/Win32/Winnt/Gina on the MSDN CD-ROM.
*/

int
WINAPI
WlxLoggedOutSAS(
    PVOID                   pWlxContext,
    DWORD                   dwSasType,
    PLUID                   pAuthenticationId,
    PSID                    pLogonSid,
    PDWORD                  pdwOptions,
    PHANDLE                 phToken,
    PWLX_MPR_NOTIFY_INFO    pMprNotifyInfo,
    PVOID *                 pProfile
    )
{
    int         result;
    PWLX_PROFILE_V1_0   pWlxProfile;
    PWSTR szProfile;
    // PMiniAccount    pAccount;
    PGlobals        pGlobals;

    pGlobals = (PGlobals) pWlxContext;

    result = pWlxFuncs->WlxDialogBoxParam(
                hGlobalWlx,
                hDllInstance,
                (LPTSTR) MAKEINTRESOURCE(IDD_LOGON_DIALOG),
                NULL,
                LogonDlgProc,
                (LPARAM) pGlobals );

    if (result == WLX_SAS_ACTION_LOGON)
    {
        result = AttemptLogon(pGlobals, pGlobals->pAccount,
                              pLogonSid, pAuthenticationId);
```

```
if (result == WLX_SAS_ACTION_LOGON)
{
    *pdwOptions = 0;
    *phToken = pGlobals->hUserToken;

    if(!GetUserProfilePath(
        pGlobals->pAccount->pszUsername,
        pGlobals->pAccount->pszDomain,
        &szProfile
        )) {
        //
        // error occurred acquiring profile path. Log error
        // here if appropriate. Default is to not provide
        // profile information as szProfile will be NULL
        // which causes *pProfile to be set to NULL
        //
    }

    //
    // if no profile is specified in the userinfo, let winlogon
    // handle locating the registry hive
    //
    if(szProfile == NULL) {
        *pProfile = NULL;
    }
    else {
        pWlxProfile = (PWLX_PROFILE_V1_0)HeapAlloc(
            GetProcessHeap(), 0, sizeof(WLX_PROFILE_V1_0) );

        if(pWlxProfile == NULL) {
            //
            // error occurred allocating memory. Log error
            // here if appropriate. Free memory associated
            // with the acquired profile path. Default is to
            // not provide profile information by supplying
            // NULL as pProfile.
            //
            HeapFree(GetProcessHeap(), 0, szProfile);
            *pProfile = NULL;
        }
        else {
            //
            // the allocation succeeded -- fill in the profile
            // information
            //
            pWlxProfile->dwType = WLX_PROFILE_TYPE_V1_0;
            pWlxProfile->pszProfile = szProfile;
            *pProfile = pWlxProfile;
        }
    }

    pMprNotifyInfo->pszUserName =
        DupString(pGlobals->pAccount->pszUsername);
    pMprNotifyInfo->pszDomain =
```

```
                DupString(pGlobals->pAccount->pszDomain);
            pMprNotifyInfo->pszPassword =
                DupString(pGlobals->pAccount->pszPassword);
            pMprNotifyInfo->pszOldPassword = NULL;
        }
    }
    return(result);
}

/*
 The following function obtains the profile path for the supplied user
 on the supplied domain.

 If the function succeeds, the return value is TRUE.
 If the function fails, the return value is FALSE, and the ProfilePath
 is set to NULL.

 If no profile path exists for the supplied user, the ProfilePath
 parameter will be set to NULL. If ProfilePath is non-NULL, the caller
 is responsible for freeing the string via HeapFree(GetProcessHeap...).

 This source code relies on the lm.h header file and the netapi32.lib
 import library.
*/
BOOL
GetUserProfilePath(
    IN LPWSTR UserName,     // UserName to retrieve profile path
    IN LPWSTR Domain,       // Domain user resides on
    OUT LPWSTR *ProfilePath // result profile path.  NULL == no profile
    )
{
    LPWSTR wTargetComputer;
    PUSER_INFO_3 ui3 = NULL;
    PUSER_MODALS_INFO_2 umi2 = NULL;
    NET_API_STATUS nas;
    BOOL bSuccess = FALSE; // assume this function will fail

    *ProfilePath = NULL;

    //
    // get the local domain name.
    // NOTE: in a gina, it would be wise to retrieve this only once,
    // in DllMain during DLL_PROCESS_ATTACH. The pointer could be
    // saved in a global variable and then compared against below.
    //
    nas=NetUserModalsGet(NULL, 2, (LPBYTE *)&umi2);
    if(nas != NO_ERROR) {
        return FALSE;
    }

    __try {

    //
    // determine if you need to look up at the domain controller
    //
```

```
if(lstrcmpiW(Domain, umi2->usrmod2_domain_name) == 0) {
    //
    // target computer is local machine
    //
    wTargetComputer = NULL;
}
else {
    //
    // target computer is the PDC computer name for the specified
    // domain
    //
    nas=NetGetDCName(NULL, Domain, (LPBYTE *)&wTargetComputer);
    if(nas != NO_ERROR) {
        __leave;
    }
}

//
// fetch the info for the user on the appropriate machine
//
nas=NetUserGetInfo(
    wTargetComputer,
    UserName,            // user name
    3,                   // info-level
    (LPBYTE *) &ui3
    );

if(nas != NO_ERROR) {
    __leave;
}

//
// if there is no profile, indicate success.
// Note that *ProfilePath will be a NULL pointer
//
if(*ui3->usri3_profile == L'\0') {
    bSuccess = TRUE;
    __leave;
}

//
// allocate storage for profile string
//
*ProfilePath = HeapAlloc(GetProcessHeap(), 0,
    (lstrlenW(ui3->usri3_profile) + 1) * sizeof(WCHAR) );

if(*ProfilePath == NULL) __leave;

//
// copy the appropriate structure memory to allocated storage
//
if(lstrcpyW(*ProfilePath, ui3->usri3_profile) != NULL)
    bSuccess = TRUE; // indicate success if the copy succeeded

} // try
```

```
    __finally {

    //
    // free the allocated buffers
    //
    if(umi2 != NULL)
        NetApiBufferFree(umi2);

    if(ui3 != NULL)
        NetApiBufferFree(ui3);

    if(wTargetComputer != NULL)
        NetApiBufferFree(wTargetComputer);

    if(!bSuccess) {
        if(*ProfilePath != NULL) {
            HeapFree(GetProcessHeap(), 0, *ProfilePath);
            *ProfilePath = NULL;
        }
    }

    } // finally

    return bSuccess;
}
```

Additional reference words: 3.50 LogonUser LsaLogonUser
KBCategory: kbprg kbhowto kbcode
KBSubcategory: BseSecurity CodeSam

How to Find Out Which Listview Column Was Right-Clicked

PSS ID Number: Q125694

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY

=======

You can use the technique described in this article to find out which column was clicked after right-clicking the listview column header.

MORE INFORMATION

================

LVN_COLUMNCLICK notifies a listview's parent window when a column is clicked using the left mouse button, but no such notification occurs when a column is clicked with the right mouse button.

Windows 95 sends an NM_RCLICK notification to the listview's parent window when a column is clicked with the right mouse button, but the message sent does not contain any information as to which column was clicked, especially if the window is sized so that the listview is scrolled to the right.

The correct way to determine which column was clicked with the right mouse button, regardless of whether the listview is scrolled, is to send the header control an HDM_HITTEST message, which returns the index of the column that was clicked in the iItem member of the HD_HITTESTINFO struct. In sending this message, make sure the point passed in the HD_HITTESTINFO structure is relative to the header control's client coordinates. Do not pass it a point relative to the listview's client coordinates; if you do, it will return an incorrect column index value.

The header control in this case turns out to be a child of the listview control of LVS_REPORT style.

The following code demonstrates this method. Note that while the code processes the NM_RCLICK notification on a WM_NOTIFY message, you also process the WM_CONTEXTMENU message, which is also received as a notification when the user clicks the right mouse button.

```
case WM_NOTIFY:
{
    if ((((LPNMHDR)lparam)->code == NM_RCLICK))
    {
       HWND hChildWnd;
       POINT pointScreen, pointLVClient, pointHeader;
       DWORD dwpos;

       dwPos = GetMessagePos();
```

```
        pointScreen.x = LOWORD (dwPos);
        pointScreen.y = HIWORD (dwPos);

        pointLVClient = pointScreen;

        // Convert the point from screen to client coordinates,
        // relative to the listview
        ScreenToClient (ghwndLV, &pointLVClient);

        // Because the header turns out to be a child of the
        // listview control, we obtain its handle here.
        hChildWnd = ChildWindowFromPoint (ghwndLV, pointLVClient);

        // NULL hChildWnd means R-CLICKED outside the listview.
        // hChildWnd == ghwndLV means listview got clicked: NOT the
        // header.
        if ((hChildWnd) && (hChildWnd != ghwndLV))
        {
           char szClass [50];

           // Verify that this window handle is indeed the header
           // control's by checking its classname.
           GetClassName (hChildWnd, szClass, 50);
           if (!lstrcmp (szClass, "SysHeader32"))
           {
              HD_HITTESTINFO hdhti;
         char szBuffer [80];

              // Transform to client coordinates
              // relative to HEADER control, NOT the listview!
              // Otherwise, incorrect column number is returned.

              pointHeader = pointScreen;
              ScreenToClient (hChildWnd, &pointHeader);

              hdhti.pt = pointHeader;
              SendMessage (hChildWnd,
                           HDM_HITTEST,
                           (WPARAM)0,
                           (LPARAM) (HD_HITTESTINFO FAR *)&hdhti);
            wsprintf (szBuffer, "Column %d got clicked.\r\n", hdhti.iItem);

              MessageBox (NULL, szBuffer, "Test", MB_OK);
           }
        }
     }
     return 0L;
}

Additional reference words: 4.00
KBCategory: kbui kbcode
KBSubcategory: UsrCtl
```

How to Find the Available Keyboard Layouts Under Windows NT

PSS ID Number: Q139571

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In some applications, you may want to find out what keyboard layouts are currently available under Windows NT, so that the application can present a list of the available keyboard layouts to the user.

The list of keyboard layouts that are currently available is in the registry:

HKEY_LOCAL_MACHINE, "system\currentcontrolset\control\keyboard layout"

You can use RegEnumKeyEx() to enumerate the subkeys (available keyboard layouts). The data value for each subkey is the file name for the keyboard layout.

MORE INFORMATION

The return value of LoadKeyboardLayout() tells you whether or not one particular keyboard layout is available. The return value is 0 when the keyboard layout is not available. However, LoadKeyboardLayout is not working correctly on Japanese Windows NT version 3.51. It returns 0 even when the requested keyboard layout is available.

Additional reference words: 4.00
KBCategory: kbprg kbhowto
KBSubcategory: wintldev

How to Find the Version Number of Win32s

PSS ID Number: Q125014

The information in this articles applies to:

- Microsoft Win32s, versions 1.0, 1.1, and 1.2

SUMMARY

=======

This article describes how to obtain the version number information for Win32s installed on a Windows 3.1 machine from one of the following places:

- From an end user perspective, on either Windows for Workgroups 3.11 or Windows NT 3.5
- A 16-bit application running on Windows 3.1
- A 32-bit application running on Windows 3.1

The following section describes in more details how to obtain this information in all the above situations.

MORE INFORMATION

================

From an End User Perspective

Because Win32s does not have a user interface, there is no obvious way to get the version number information for Win32s that is installed on Windows 3.1. However, end users have the following two options:

- Read the <windir>\SYSTEM\WIN32S.INI file, which has an entry for version information. Because this .INI file can be updated by the Setup program of any Win32s application, this information is not completely reliable.

- From File Manager on Windows for Workgroups 3.11 or Windows NT 3.5, select the WIN32S16.DLL and choose Properties from the File menu. This method yields a dialog box with version information on Win32s. Remember that WIN32S16.DLL is a 16-bit DLL; however, File Manager on Windows NT 3.5 can still read this version resource information.

From a 16-Bit Application

To get version number information for Win32s from a 16-bit application, use the Win32s specific function, GetWin32sInfo(), which is documented in the Win32s Programmer's Reference. This function is exported by the 16-bit W32SYS.DLL file in Win32s 1.1 and later. The GetWin32sInfo() function fills a specified structure with the information from Win32s VxD. Usually a 16-bit Windows setup program should use this function to determine if Win32s is already installed before continuing installation. Note that a 16-bit program must use LoadLibrary and GetProcAddress to call the function because the function did not exist in Win32s version 1.0.

The following example on using GetWin32sInfo() is extracted from the Win32s Programmer's Reference:

```
// Example of a 16-bit application that indicates whether Win32s is
// installed, and the version number if Win32s is loaded and VxD is
// functional.

BOOL FAR PASCAL IsWin32sLoaded(LPSTR szVersion)
{
    BOOL              fWin32sLoaded = FALSE;
    FARPROC           pfnInfo;
    HANDLE            hWin32sys;
    WIN32SINFO        Info;

    hWin32sys = LoadLibrary("W32SYS.DLL");

    if (hWin32sys > HINSTANCE_ERROR) {
        pfnInfo = GetProcAddress(hWin32sys, "GETWIN32SINFO");
        if (pfnInfo) {
            // Win32s version 1.1 is installed
            if (!(*pfnInfo)((LPWIN32SINFO) &Info)) {

                fWin32sLoaded = TRUE;
                wsprintf(szVersion, "%d.%d.%d.0",
                         Info.bMajor, Info.bMinor, Info.wBuildNumber);
            } else
                fWin32sLoaded = FALSE;    // Win32s VxD not loaded.
        } else {
            // Win32s version 1.0 is installed.
            fWin32sLoaded = TRUE;
            lstrcpy( szVersion, "1.0.0.0" );
        }
        FreeLibrary( hWin32sys );
    } else                                // Win32s not installed.
        fWin32sLoaded = FALSE;


    return fWin32sLoaded;
}
```

From a 32-Bit Application

To determine if Win32s is installed, use the function GetVersion(); to then get the version of Win32s use the function, GetVersionEx(). This function fills a specified structure with version information of Win32s on Windows 3.1. The following is an example illustrating the use of this function:

```
// Example of a 32-bit code that determines the operating system installed
// and the version number on all platforms: Windows NT, Windows 95, Win32s.

typedef BOOL (*LPFNGETVERSIONEX) (LPOSVERSIONINFO);

BOOL IsWin32sLoaded(char *szVersion)
```

```
{
  BOOL              fWin32sLoaded = FALSE;
  DWORD             dwGetVer;
  HMODULE           hKernel32;
  OSVERSIONINFO     ver;
  LPFNGETVERSIONEX    lpfnGetVersionEx;

  // First, check if Win32s is installed
  dwGetVer = GetVersion();
  if (!(dwGetVer & 0x80000000))
  {
     //  Windows NT is loaded
     // Note, GetVersion will also return version number on Windows NT

     return;
  }
  else if (LOBYTE(LOWORD(dwVersion))< 4)
  {
    // Win32s is loaded
    fWin32sLoaded = TRUE;
  }
  else {
     // Windows 95 is loaded
     // Note, GetVersion will also return version number on Windows 95

     return;
  }

  // Now, let's find the version number of Win32s
    hKernel32 = GetModuleHandle("Kernel32");
    if (hKernel32)
    {
        lpfnGetVersionEx = (LPFNGETVERSIONEX)GetProcAddress(hKernel32,
"GetVersionExA");
        if (lpfnGetVersionEx)
   {
           // Win32s version 1.15 or later is installed
           ver.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
           if (!(*lpfnGetVersionEx)((LPOSVERSIONINFO) &ver))
             DisplayError("GetVersionEx");
           else
             wsprintf(szVersion, "%d.%d.%d - %s", ver.dwMajorVersion,
ver.dwMinorVersion,
                      ver.dwBuildNumber, PlatformName(ver.dwPlatformId));
   }
   else
   {
// This failure could mean several things
// 1. On an NT system, it indicates NT version 3.1 because GetVersionEx()
//    is only implemented on NT 3.5.
// 2. On Windows 3.1 system, it means either Win32s version 1.1 or 1.0 is
//    installed. You can distinguish between 1.1 and 1.0 in two ways:
// a. Get version info from WIN32S16.DLL like File Manager on NT does.
// b. Thunk to 16-bit side and call GetWin32sInfo.
```

```
   }
    }

 return (fWin32sLoaded);
}
```

NOTE: In general, 32-bit applications that use Win32s should always ship with the latest version of Win32s. Therefore the detection code above can be greatly simplified if determination of previous versions of Win32s is not needed.

Additional reference words: 3.10 win32s w32s win32 wfw
KBCategory: kbenv
KBSubcategory: W32s

How to Force a ScreenSaver to Close Once Started in Windows NT

PSS ID Number: Q140723

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows NT versions 3.1, 3.5, 3.51

SUMMARY

Sometimes applications need to terminate a screensaver that is already running. In Windows 3.1 or Windows 95, a screensaver could be terminated easily by posting a WM_CLOSE message to the currently active screensaver window as in this example:

```
    PostMessage (GetActiveWindow(), WM_CLOSE, 0, 0L);
```

Windows NT, however, introduces the concept of separate desktops, so that applications can run on one, and screen savers can run on another. This makes screensaver termination under Windows NT a bit more difficult.

MORE INFORMATION

Under Windows NT, obtaining a handle to the currently active screensaver window is not as straightforward as it is in Windows 3.1 and Windows 95. Calling GetForegroundWindow() returns NULL because the screensaver is running on a different desktop than the calling application. Similarly, calling FindWindow ("WindowsScreenSaverClass", NULL) to determine if the screen saver is currently active won't work either.

The way to do it is to get a handle to the screen saver's desktop, enumerate that desktop's windows, and then post a WM_CLOSE to the screen saver window.

The following code demonstrates how to do this. Note that if a screen saver password is set, the following code brings up the password dialog box, prompts the user for a password, and then actually terminating the screen saver application.

```
   BOOL CALLBACK KillScreenSaverFunc(HWND hwnd, LPARAM lParam)
   {
      PostMessage(hwnd, WM_CLOSE, 0, 0);
      return TRUE;
   }

   HDESK hdesk;

   hdesk = OpenDesktop(TEXT("Screen-saver"),
                       0,
                       FALSE,
                       DESKTOP_READOBJECTS | DESKTOP_WRITEOBJECTS);
```

```
   if (hdesk)
   {
      EnumDesktopWindows(hdesk, KillScreenSaverFunc, 0);
      CloseDesktop(hdesk);
   }
```

Note that terminating a screensaver that is already running as demonstrated above is totally separate from disabling the screen saver altogether, so that no screen saver starts after the designated time period expires. This can be done easily using:

```
   SystemParametersInfo( SPI_SETSCREENSAVEACTIVE,
                         FALSE,
                         0,
                         SPIF_SENDWININICHANGE
                       );
```

This method works well for terminating the currently running screen saver. However, one problem that you might encounter is that the system will not restart the screen saver unless the user moves the mouse or presses a key. If you need the screen saver to start up again, you'll need to reinitialize the time-out period. Do this by:

- Calling SystemParametersInfo(SPI_SETSCREENSAVEACTIVE, TRUE, 0, SPIF_SENDWININICHANGE).

  -or-

- Using SetCursorPos() to simulate user input.

Both of these methods will cause the system to restart the time-out counter for the screen saver.

Additional reference words: 3.10 3.50 deactivate disable stop running turn off
KBCategory: kbgraphic kbhowto kbcode
KBSubcategory: GdiScrSav

How to Get and Set the Default Printer in Windows

PSS ID Number: Q135387

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In all versions of Windows, the appropriate way to get the default printer is to use GetProfileString, and the appropriate way to set the default printer is to use WriteProfileString. This works whether the default printer information is stored in the WIN.INI file or in the registry.

MORE INFORMATION

Notes to Keep in Mind

- The Device value you get or set actually contains three elements separated by commas as follows:

  <printer name>,<driver name>,<port>

  For example:

  My Printer,HPPCL5MS,lpt1:

- When setting the default printer, you must specify valid names for these elements. That is, you must specify a valid printer, driver, and port. If not, programs such as Print Manager may set the printer back to the previous valid printer, and other programs may become very confused.
- Windows 95 and Windows NT map most .INI file references to the registry. Because of this mapping, GetProfileString and WriteProfileString still function as they do under 16-bit Windows (Microsoft Windows and Windows for Workgroups).
- After setting the default printer, notify all other applications of the change by broadcasting the WM_WININICHANGE message.

Sample Code

Under Windows NT, the default printer does not change based on this code,

this code will change only the default printer for this process. Another application will still default to the original printer driver set as the default.

```
// This code uses a sample profile string of "My Printer,HPPCL5MS,lpt1:"
// To get the default printer for Windows 3.1, Windows 3.11,
// Windows 95, and Windows NT use:
GetProfileString("windows", "device", ",,,", buffer, sizeof(buffer));
```

```
// To set the default printer for Windows 3.1 and Windows 3.11 use:
WriteProfileString("windows", "device", "My Printer,HPPCL5MS,lpt1:");
SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, 0L);
```

```
// To set the default printer for Windows 95 use:
WriteProfileString("windows", "device", "My Printer,HPPCL5MS,lpt1:");
SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0L,
(LPARAM)(LPCTSTR)"windows");
```

```
// To set the default printer for Windows NT use:
/* Note printer driver is usually WINSPOOL under Windows NT */
WriteProfileString("windows", "device", "My Printer,WINSPOOL,lpt1:");
SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0L, 0L);
```

There are two circumstances where the code won't work:

1. If the customer leaves out the SendMessage, no other application will recognize the change in the .INI settings.

2. If a different 32-bit application does not handle the WIN.INI change message, then it will appear to that application as if the default printer has not been changed. The user will need to exit and re-enter Windows 95 to have the other application recognize the printer change.

This is the preferred method of changing the printer if the code is to be platform independent; this method will work on Windows 3.1, Windows 95 and Windows NT.

For additional information, please see the following article(s) in the Microsoft Knowledge Base:

   ARTICLE-ID: Q140560
   TITLE   : How to Set the Default Printer Programmatically in Windows 95

Additional reference words: 3.10 4.00 3.50 3.11 3.51
KBCategory: kbprint kbcode
KBSubcategory: GdiPrn

How to Get Message Text from Networking Error Codes

PSS ID Number: Q149409

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  for Windows NT version 3.51

SUMMARY

In Windows NT, it is sometimes necessary to display error text associated with error return codes returned from networking related API functions. For example, this is needed with the family of functions that may return networking specific error codes in the Windows NT Lan Manager API set.

The error text for these messages is found in the message table file named Netmsg.dll, which is found in %systemroot%\system32. This file contains error messages in the range NERR_BASE (2100) through MAX_NERR (NERR_BASE+899). These error codes are defined in the Windows NT Lan Manager header file Lmerr.h.

MORE INFORMATION

The LoadLibrary() and LoadLibraryEx() Win32 API functions can be used to load Netmsg.dll. The FormatMessage() Win32 API can be used to map an error code to message text given a module handle to the Netmsg.dll file.

Sample Code

```
/*
    The following sample illustrates how to display error text
    associated with Networking related error codes, in addition
    to displaying error text associated with system related error
    codes.

    This sample relies on the following import library:
    User32.lib

*/

#include <windows.h>
#include <stdio.h>

#include <lmerr.h>

void
DisplayErrorText(
    DWORD dwLastError
    );
```

```
int
__cdecl
main(
    void
    )
{
    //
    // display a networking related error string
    //
    printf("Network related error string follows:\n");
    DisplayErrorText(2226);

    //
    // display a system related error string
    //
    printf("\nSystem related error string follows:\n");
    SetLastError(ERROR_FILE_NOT_FOUND);
    DisplayErrorText(GetLastError());

    return 0;
}

void
DisplayErrorText(
    DWORD dwLastError
    )
{
    HMODULE hModule = NULL; // default to system source
    LPSTR MessageBuffer;
    DWORD dwBufferLength;

    //
    // if dwLastError is in the network range, load the message source
    //
    if(dwLastError >= NERR_BASE && dwLastError <= MAX_NERR) {
        hModule = LoadLibraryEx(
            TEXT("netmsg.dll"),
            NULL,
            LOAD_LIBRARY_AS_DATAFILE
            );
    }

    //
    // call FormatMessage() to allow for message text to be acquired
    // from the system or the supplied module handle
    //
    if(dwBufferLength = FormatMessageA(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_IGNORE_INSERTS |
        FORMAT_MESSAGE_FROM_SYSTEM | // always consider system table
        ((hModule != NULL) ? FORMAT_MESSAGE_FROM_HMODULE : 0),
        hModule, // module to get message from (NULL == system)
        dwLastError,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // default language
        (LPSTR) &MessageBuffer,
```

```
        0,
        NULL
        ))
    {
        DWORD dwBytesWritten;

        //
        // Output message string on stderr
        //
        WriteFile(
            GetStdHandle(STD_ERROR_HANDLE),
            MessageBuffer,
            dwBufferLength,
            &dwBytesWritten,
            NULL
            );

        //
        // free the buffer allocated by the system
        //
        LocalFree(MessageBuffer);
    }

    //
    // if you loaded a message source, unload it
    //
    if(hModule != NULL)
        FreeLibrary(hModule);
}
```

Additional reference words: error string LanMan
KBCategory: kbprg kbhowto kbcode
KBSubcategory: NtwkMisc BseMisc CodeSam

How to Get Smallest Bounding Rectangle for Drawing Functions

PSS ID Number: Q139217

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Windows maintains an accumulated bounding rectangle for each application. An application can retrieve this bounding rectangle by calling GetBoundsRect, and it can set this rectangle by calling SetBoundsRect. But this rectangle may not be the smallest bounding rectangle if your drawing functions contain functions doing text output such as TextOut. GDI does not calculate ahead of time how far across the TextOut functions are going to draw. GDI just sets the right side of the bounding rectangle to a large number.

MORE INFORMATION
================

To get an accurate and smallest bounding rectangle, call LockWindowUpdate and pass the handle of the window to which you are going to draw. Then Windows records the extent of any attempted operations in a bounding rectangle without doing the actual drawing. When the window is unlocked by LockWindowUpdate(NULL), you can either use GetUpdateRect or intercept the rcPaint member of the PAINTSTRUCT in your WM_PAINT message handler to retrieve this smallest bounding rectangle.

Additional reference words: 3.50 4.00 GetUpdateRect
KBCategory: kbgrahpic kbhowto
KBSubcategory: GdiDrw GdiMisc

How to Get Windows NT PolyDraw() Functionality in Windows 95

PSS ID Number: Q135059

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

This article shows by example how to get the functionality provided by the Win32 function PolyDraw() in Windows 95.

MORE INFORMATION

The Windows NT PolyDraw() function draws a set of line segments and Bezier curves. PolyDraw() can be used in place of consecutive calls to the MoveToEx(), LineTo(), and PolyBezierTo() functions to draw disjoint figures. The lines and curves are drawn using the current pen, and figures are not filled. If there is an active path started by calling BeginPath(), then PolyDraw() adds to the path. The points contained in the lppt array and in the lpbTypes array indicate whether each point is part of a MoveToEx(), LineTo(), or PolyBezierTo() operation. It is also possible to close figures.

Code Sample

The following function enables you to get the functionality of the Windows NT PolyDraw() function in Windows 95:

```
/*************************************************************************
 *                                                                       *
 * FUNCTION:  PolyDraw95(HDC, LPPOINT, LPBYTE, int)                      *
 *                                                                       *
 * PURPOSE:   Draws the points returned from a call to GetPath()         *
 *            to an HDC                                                  *
 *                                                                       *
 * NOTES:     This function is similar to the Windows NT PolyDraw        *
 *            function, which draws a set of line segments and Bezier    *
 *            curves. Because PolyDraw is not supported in Windows 95    *
 *            this PolyDraw95 function is used instead.                  *
 *                                                                       *
 *************************************************************************/
BOOL PolyDraw95(HDC  hdc,              // handle of a device context
                CONST LPPOINT lppt,      // array of points
                CONST LPBYTE lpbTypes, // line and curve identifiers
                int  cCount)             // count of points
{
  int i;
```

```
  for (i=0; i<cCount; i++)
    switch (lpbTypes[i]) {
      case PT_MOVETO :
         MoveToEx(hdc, lppt[i].x, lppt[i].y, NULL);
         break;

      case PT_LINETO | PT_CLOSEFIGURE:
      case PT_LINETO :
         LineTo(hdc, lppt[i].x, lppt[i].y);
         break;

      case PT_BEZIERTO | PT_CLOSEFIGURE:
      case PT_BEZIERTO :
        PolyBezierTo(hdc, &lppt[i], 3);
       i+=2;
         break;
    }

   return TRUE;
}
```

Additional reference words: 95 4.00 PolyDraw Draw Poly Stones Win95 GDI
KBCategory: kbgraphic kbcode
KBSubcategory: GdiMisc

How to Gracefully Fail at Service Start

PSS ID Number: Q115829

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

=======

If an error occurs while your service is running or initializing (SERVICE_START_PENDING) and you need to stop the service process, do the following:

1. Clean up any resources that are being used (threads, memory, and so forth). You should start sending a SERVICE_STOP_PENDING status if the clean up process is lengthy. Be sure to update the Service Control Manager as demonstrated in the Win32 SDK SERVICE sample.

2. Send out a SERVICE_STOPPED status from the last thread to terminate before it calls ExitThread().

3. Set SERVICE_STATUS.dwWin32ExitCode and/or SERVICE_STATUS.dwServiceSpecificExitCode to values that indicate why the service is stopping. If you return a value for the dwServiceSpecificErrorCode field, then the dwWin32ExitCode field should be set to ERROR_SERVICE_SPECIFIC_ERROR.

   The reason for setting these values is that if a service fails its operation, but returns an exit code of 0, the following error message is returned by default:

      Error 2140: An internal Windows NT error occurred

MORE INFORMATION

================

When the last service in the process has terminated (you may have multiple services in the service process), the StartServiceCtrlDispatcher() call in the main thread returns. The main routine should call ExitProcess() because all of the services have terminated.

REFERENCES

==========

There is a termination sample in the "Win32 Programmer's Reference," in the "Services" overview section (58.2.2), "Writing a ServiceMain Function." This is a simple situation where the service process only consists of one thread. This thread returns when it is ready to terminate, instead of calling ExitThread().

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseService

How to Handle Data Types Correctly with the Thunk Compiler

PSS ID Number: Q142564

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95, version 4.0

SUMMARY

The thunk compiler supports most of the built-in and user-defined types that the C or C++ languages support. It also supports a subset of the calling conventions that the Microsoft Visual C++ compiler supports. To handle situations where you need to pass or return data of an unsupported type, you need to massage the data into a type that the thunk compiler does recognize. This article explains how to do so for commonly used types supported by the C and C++ languages but not by the thunk compiler.

MORE INFORMATION

There are several strategies for handling data types and function calling conventions that the thunk compiler does not support. These strategies generally rely on substitution and pre- or post-processing the data, and require an understanding of how the thunk compiler translates data. The thunk compiler uses the following rules to translate data:

- Automatically translate 32-bit pointers into 16-bit far pointers and vice-versa.

- Pass signed and unsigned scalar types (varieties of char, int, and long) unmodified with the exception of int. Because the type int is 32 bits wide in 32-bit code and 16 bits wide in 16-bit code, truncate int values on the 16-bit side, and sign-extend them on the 32-bit side.

These two translation rules make thunks handle parameters as expected in C or C++. A pointer is a pointer on both sides, and a scalar value is the same on both sides too. These rules come in handy when you need to use the Generic Thunk API functions in combination with flat thunks.

Before delving into how to handle specific data types, you need to consider the general principles for substituting function calling conventions and data types. These are simple:

- Write wrapper functions around the thunk to handle unsupported calling conventions.

- Substitute supported types for unsupported types in the thunk scripts. The supported type must be the same length as the unsupported type, and generally should not be a pointer type.

- If the unsupported type is larger than eight bytes, then it usually

   should be handled by passing a pointer to it and then manually
   interpreting it on the other side of the thunk. Unions and arrays of
   structures fall into this category.

 - To handle unsupported types in return values, write wrapper functions
   for both sides of the thunk and where necessary, give the actual thunk
   function an extra parameter that points to a dynamically-allocated
   buffer.

Function Calling Conventions

The Windows 95 thunk compiler supports a subset of the calling conventions supported by the Microsoft Visual C++ compiler. Only __stdcall is supported on the 32-bit side of a thunk; on the 16-bit side, only __pascal is supported. __cdecl and __fastcall are not supported calling conventions for either the 16-bit or 32-bit sides of the thunk. Although the thunk must use __pascal and __stdcall calling conventions, the thunk may call or be called by other functions that use any calling convention.

To thunk to a function that uses the __cdecl or __fastcall calling convention, you must create your thunk function with the supported calling conventions and have the target side of the thunk call the real function. Here is an example that shows how a Win32 application might call a 16-bit __cdecl function:

```
/*-------------------------------------------------------------
Code inside Win32 application or DLL
-------------------------------------------------------------*/

/*
   Declaration for benefit of the 32-bit side of thunk, note
   that in Win32 programs, PASCAL is defined as __stdcall,
   and FAR is defined as nothing.
*/
void FAR PASCAL C_Function16 (int x);

/*
   MyCFunction is a C calling convention function to interface
   with rest of the Win32 application.  It calls the thunk,
   which is __stdcall.
*/
void MyCFunction (int x)
{
   C_Function16 (x);
}

/*-------------------------------------------------------------
Code inside Win16 DLL (target of thunk)
-------------------------------------------------------------*/

// Declaration for the real C function
void Real_C_Function (int x);

/*
```

```
   C_Function16 is the 16-bit side of the thunk.  In a 32->16
   thunk, it is the target.  Here, it calls the real C function
   that the Win32 application wanted to call in the first
   place. The FAR macro and the __export keyword are used to
   make sure the function is exported correctly from the 16-bit
   DLL.
*/
void FAR PASCAL __export C_Function16 (int x)
{
   Real_C_Function (x);
}


/*------------------------------------------------------------
Thunk script
------------------------------------------------------------*/
enablemapdirect3216 = true;

void C_Function16 (int x)
{
}
```

If you try to build a thunk and don't use the __stdcall or __pascal calling conventions, you will get "unresolved external" errors from the linker when building because the function names produced by the C/C++ compiler don't match the names in the code produced by the thunk compiler.

Return Values

The thunk compiler does not support all types as return values that the C and C++ languages support. The limitations stem from the problems of translating addresses from 32-bit address spaces to 16-bit address spaces and vice-versa.

The thunk compiler supports returning all integral types. The signed and unsigned varieties of char, short int, and long int are the same size on both sides of a thunk and thus are returned unmodified. The values of the int type will be truncated to 16-bits upon return from a 16->32 thunk, and will be sign-extended to 32-bits on return from a 32->16 thunk.

Structures may be returned if they do not require repacking. A structure does not require repacking if the types of all members of the structure are the same size in 16-bit and 32-bit code and the structure is packed with the same alignment on both sides of the thunk. Use #pragma pack to pack the structure with the same value in the 32-bit and 16-bit source code, and specify the same packing on the thunk compiler command line.

Thunks from 16-bit code to 32-bit code cannot return pointers because the context of the 32-bit address space is not global. The 32-bit address space for the Win32 target DLL is mapped between 4 megabytes and 2 gigabytes, and it is context-switched as are all other Win32 process address spaces. This means that a Win16 application or DLL could not use a pointer returned by a 16->32 thunk without causing a general protection (GP) fault. You can work around this limitation by writing a wrapper function for the 16-bit side of

the thunk that allocates memory and passes it to the thunk function as an extra parameter. The 32-bit side of the thunk should also be a wrapper around the real function to handle the extra parameter and copy the data to the memory allocated by the 16-bit wrapper function.

Pointers are supported as return types in 32->16 thunks as long as the base type requires the same number of bytes on both sides of the thunk and does not require repacking. If the size of the base type of a pointer differs between the Win16 and Win32 sides, the thunk compiler generates an error. For example, a char is one byte on both sides of the thunk, and thus a pointer to a char is supported as a return type. A pointer to an int cannot be returned because an int is 16 bits on the 16-bit side of the thunk, but 32 bits on the 32-bit side. A pointer to a structure may be returned if the structure does not need to be repacked.

To return a pointer to a data type whose size is different on the 16-bit and 32-bit sides of a 32->16 thunk, you should write a wrapper function that returns the real type, and have the wrapper pass a buffer to the thunk function as an extra parameter. The target side of the thunk should put the return value into this buffer, and then the wrapper on the calling side should return the data from the buffer once the target side of the thunk returns.

The thunk compiler has one special type (bool) that is not found in C or C++ compilers. You should use this type in your thunk scripts for functions that return boolean values when the TRUE result is any non-zero value.

Floating-Point Data Types

The thunk compiler does not support any floating point types. There are three floating point types to consider, and how you approach them depends on which you need to use.

A float is four bytes long on both the 16-bit and 32-bit sides of a thunk. A float can be passed to a thunk function by declaring it as a DWORD (unsigned long) in the thunk script. This makes the thunk compiler pass a four byte value without translating its value.

A double is eight bytes long on both the 16-bit and 32-bit sides of the thunk. To pass a double to a thunk function, declare a struct containing two DWORDs (unsigned longs) in the thunk script, and pass the struct into the function in the thunk script. Here is an example 32->16 thunk:

```
/*-------------------------------------------------------------
Code on Win32 side of thunk
-------------------------------------------------------------*/

/*
   Declaration for the benefit of the 32-bit side of thunk,
   note that in Win32 programs, PASCAL is defined as __stdcall,
   and FAR is defined as nothing.
*/
double FAR PASCAL sqr16 (double x);
```

```
double sqr32 (double x)
{
   return(sqr16 (x));
}


/*-------------------------------------------------------------
Code inside Win16 DLL (target of thunk)
-------------------------------------------------------------*/

/*
   sqr16() is the 16-bit side of the thunk. In a 32->16 thunk,
   it is the target. The FAR macro and the __export keyword
   are used to make sure it is exported correctly from the 16-bit
   DLL.
*/
double FAR PASCAL __export sqr16 (double x)
{
   return (x * x);
}


/*-------------------------------------------------------------
Thunk script
-------------------------------------------------------------*/
typedef unsigned long DWORD;
typedef struct _MYDOUBLE
{
   DWORD dwLow;
   DWORD dwHigh;
} MYDOUBLE;

enablemapdirect3216 = true;

MYDOUBLE sqr16 (MYDOUBLE x)
{
}
```

In 16-bit code, a long double is a native 80-bit type of the floating point processors in the Intel x86 microprocessor family. Because RISC processors do not have a native 80-bit floating point type, C and C++ compilers for Win32 platforms such as Microsoft Visual C++ implement this type as a 64-bit double. If you cannot tolerate the loss of precision of converting 80-bit long doubles to 64-bit doubles, then you should pass the 80-bit value as a structure consisting of two DWORDs (unsigned longs) and a WORD (unsigned short), and you will have to handle the value manually on the 32-bit side of the thunk.

Pointers

The thunk compiler automatically translates pointers to most types. It properly handles pointers to the following data types:

- All scalar data types
- Structures
- Pointers within structures if the object pointed to does not require repacking (such as the int type).

Pointers to pointers are translated partially. The "outside" pointer is translated, but the pointer it points to -- the "inside" pointer-- is not translated.

The thunk compiler does not handle all cases of pointers used in aggregate types, such as arrays of pointers.

Finally, the thunk compiler places limitations on pointers as return values as described in the "Return Values" section earlier in this article.

There may be times when you want to pass an address without having it translated. Because the thunk compiler automatically translates pointers, you should use the DWORD type instead of a pointer type to pass addresses that you don't want translated. Then, you can use the untranslated address on the target side of the thunk, possibly in calls to the Generic Thunk APIs. However, keep in mind that addresses that have not been translated will cause an access violation if dereferenced on the target side of the thunk.

Unions and C++ Classes

Unions are not supported by the thunk compiler because it cannot determine which type the union will actually hold at run time. For example, the following union must be treated differently by the thunk compiler depending on which member the application was actually using when it made the thunk call:

```
union
{
   DWORD dwIntegerValue;
   LPSTR szFileName;
}
```

The problem the thunk compiler has with this union is whether it should generate code to pass the value unchanged, assuming that the dwIntegerValue member will be used, or to translate the value as a pointer, assuming that szFileName will be used. Because the thunk compiler cannot make this determination, it cannot generate the correct code to handle this union. You should handle union types in thunks by declaring a structure in the thunk script large enough to hold the union, and then handle the union's data manually on both sides of the thunk.

Do not thunk C++ objects. Objects are not supported by the thunk compiler because there isn't any way to reliably pass them as parameters to thunks. In addition to specifying the object's physical layout, a class may also define a vtable which contains addresses of all the class's virtual functions. Because all objects of a class share a single vtable, there is no way to translate the vtable so that it can be used by some objects that have been thunked and others that haven't.

Linked Lists and other Dynamic Data Structures

Although the thunk compiler supports structures with pointers, it does not provide a way to handle linked lists and other dynamic structures. This is because the thunk compiler must know the size of the data it translates at compile time, but the number of elements that dynamically-allocated structures contain is indeterminate at compile time.

REFERENCES

Thunk Compiler reference in the Microsoft Win32 SDK Documentation
Generic Thunk API reference in the Microsoft Win32 SDK Documentation

Additional reference words: 4.00 GPF
KBCategory: kbprg kbhowto kbcode
KBSubcategory: BseMisc

How to Handle FNERR_BUFFERTOOSMALL in Windows 95

PSS ID Number: Q131462

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SUMMARY

When an application uses the Open File common dialog with the OFN_ALLOWMULTISELECT flag, there is a danger that the buffer passed to the common dialog in the OPENFILENAME.lpstrFile field will be too small. In this situation, GetOpenFileName() will return an error value and CommDlgExtendedError() will return FNERR_BUFFERTOOSMALL.

To work around this problem, watch for the Open or OK button to be pressed in the dialog hook; then reallocate the buffer if necessary.

This technique works on Windows version 3.1, Windows NT, and Windows 95, but the implementation details are different when dealing with Windows 95 Explorer-type dialog boxes versus traditional Open and Save common dialog boxes. This article explains how to do it in Windows 95.

MORE INFORMATION

With the introduction of the new common dialogs for Windows 95, a new way of handling the FNERR_BUFFERTOOSMALL error was developed. It is still necessary to watch for the Open button to be pressed and reallocate the buffer if needed, but the way to watch for the OK is much different.

When you install a hook on the Open File common dialog in Windows 95 using the OPENFILENAME.lpfnHook member, the dialog you are hooking is a child of the main Open File dialog. Therefore, to intercept the OK button, you need to subclass the parent dialog. To do this, you can install the hook procedure and watch for the CDN_INITDONE notification. The Open File dialog will send this as part of a WM_NOTIFY message when the initialization for the dialog is complete. For example:

```
LRESULT CALLBACK DialogHook(HWND hwnd, UINT uMsg, WPARAM wParam,
                            LPARAM lParam)
{
    static HWND hwndParentDialog;
    LPOFNOTIFY lpofn;

    switch (uMsg)
    {
        case WM_INITDIALOG:
            // You need to use a copy of the OPENFILENAME struct used to
            // create this dialog. You can store a pointer to the
```

```
            // OPENFILENAME struct in the ofn.lCustData so you can retrieve
            // it here in the lParam. Once you have it, you need to hang on
            // to it. Using window properties provides a good thread safe
            // solution to using a global variable.

            SetProp(hwnd, "OFN", lParam);
            return (0);

        case WM_NOTIFY:
            // The OFNOTIFY struct is passed in the lParam of this message.

            lpofn = (LPOFNOTIFY) lParam;

            switch (lpofn->hdr.code)
            {
                CDN_INITDONE:
                    // Subclass the parent dialog to watch for the OK
                    // button.

                    hwndParentDialog = GetParent(hwnd);
                    g_lpfnDialogProc =
                            (FARPROC) SetWindowLong(hwndParentDialog,
                                                    DWL_DLGPROC,
                                                    OpenFileSubclassProc);
                    break;

            }
            return (0);

        case WM_DESTROY:
            // Need to clean up the subclassing we did on the dialog.
            SetWindowLong(hwndParentDialog, DWL_DLGPROC, g_lpfnDialogProc);

            // Also need to free the property with the OPENFILENAME struct
            RemoveProp(hwnd, "OFN");
            return (0);
    }
    return (0);
}
```

Once the parent dialog is subclassed, the program can watch for the actual Open button. When the program gets the Open button command, it needs to check to see if the buffer originally allocated is large enough to handle all the files selected. The CommDlg_OpenSave_GetFilePath() API will return the length needed. Here is an example of the subclass procedure:

```
LRESULT CALLBACK OpenFileSubclassProc(HWND hwnd, UINT uMsg, WPARAM wParam,
                                      LPARAM lParam)
{
    LPTSTR lpsz;
    WORD   cbLength;

    switch (uMsg)
    {
        case WM_COMMAND:
```

```
            switch (LOWORD(wParam))
            {
                case IDOK:
                    // Need to verify the original buffer size is large
                    // enough to handle the files selected. The
                    // CommDlg_OpenSave_GetFilePath() API will return the
                    // length needed for this buffer.

                    cbLength = CommDlg_OpenSave_GetFilePath(hwnd, NULL, 0);

                    // OFN_BUFFER_SIZE is the size of the buffer originally
                    // used in the OPENFILENAME.lpszFile member.

                    if (OFN_BUFFER_SIZE < cbLength)
                    {
                        // The buffer is too small, so allocate a
                        // new buffer.
                        lpsz = (LPTSTR) HeapAlloc(GetProcessHeap(),
                                                  HEAP_ZERO_MEMORY,
                                                  cbLength);
                        if (lpsz)
                        {
                            // The OFN struct is stored in a property of
                            // the dialog window.

                            lpofn = (LPOPENFILENAME) GetProp(hwnd, "OFN");

                            lpofn->lpstrFile = lpsz;
                            lpofn->nMaxFile  = cbLength;
                        }
                    }

                    // Now let the dialog handle the message normally.
                    break;
            }
            break;
    }

    return (CallWindowProc(g_lpfnDialogProc, hwnd, uMsg, wParam, lParam));
}
```

The dialog should now return without error. Be aware that the buffer allocated in the subclass procedure needs to be freed once the dialog returns.

Finally, this technique only works for 32-bit applications that are using the Explorer-type common dialogs. For 32-bit applications that don't use the OFN_EXPLORER flag, Windows 95 thunks to the 16-bit common dialog and the hook function only gets a copy of the OPENFILENAME structure.

Additional reference words: 4.00
KBCategory: kbui kbcode
KBSubcategory: UsrCmnDlg

How to Ignore WM_MOUSEACTIVATE Message for an MDI Window

PSS ID Number: Q62068

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

In order to make an MDI window to become active and have the caret be in the same position as when the window was last active you need to process the WM_MOUSEACTIVATE message and return MA_ACTIVATEANDEAT for the first time. Therefore, you need to set a Boolean flag in the WM_MDIACTIVATE message so that the return is set only once. The sample code below can be used to modify the MULTIPAD sample application. Also, the following is documentation on MA_ACTIVATE* messages, taken from the Windows 3.0 final SDK README.WRI file:

```
   WM_MOUSEACTIVATE

   Return Value    The return value specifies whether the window
                   should be activated and whether the mouse event
                   should be discarded. It must be one of the
                   following values:

                   Value                   Meaning
                   -----                   -------

                   MA_ACTIVATE             Activate the window.
                   MA_NOACTIVATE           Do not activate the window.
                   MA_ACTIVATEANDEAT       Activate the window and
                                           discard the mouse event.
```

SAMPLE CODE

```
/* --- multipad.c  MPMDIWndProc section --- */

 case WM_MOUSEACTIVATE:   // added
     if (bEatMessage) {
        bEatMessage = FALSE;
        return (LONG)MA_ACTIVATEANDEAT ;
     }
     /* else break */
     break;

 case WM_MDIACTIVATE:
     /* If we're activating this child, remember it */
     if (wParam){
       hwndActive     = hwnd;
```

```
        hwndActiveEdit = (HWND)GetWindowWord (hwnd, GWW_HWNDEDIT);
        bEatMessage = TRUE;      // added
      }
      else{
            hwndActive     = NULL;
            hwndActiveEdit = NULL;
      }
      break;
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMdi

How to Implement a Recursive RegDeleteKey for Windows NT

PSS ID Number: Q142491

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Microsoft Windows NT version 3.51

SUMMARY

In Windows 95, the RegDeleteKey function not only deletes the particular key specified but also any subkey descendants. In contrast, the Windows NT version of this function deletes only the particular key specified and will not delete any key that has subkey descendants.

To delete a key and all of its subkeys in Windows NT, a recursive delete function is implemented using RegEnumKeyEx and RegDeleteKey. This recursive delete function works by: (1) traversing down each subkey branch, one branch at a time, enumerating each key at each subkey level, until the last subkey leaf is reached and (2) individually deleting each subkey in reverse succession, one branch at a time, until even the particular key specified is deleted.

MORE INFORMATION

Starting at the particular key specified, each key is traversed by using RegEnumKeyEx, which determines if there are any subkeys. If so, the subkey's name is passed to the recursive delete function in order to traverse to the next subkey. This process is repeated for all subkey descendants. When RegEnumKeyEx reports that there are no more subkeys (that is, ERROR_NO_MORE_ITEMS) for the current key, a subkey leaf has been reached.

Once the subkey leaf is deleted using RegDeleteKey, the recursive delete function re-examines the parent key for any remaining subkeys. If a subkey does exist, it is also traversed until a subkey leaf is reached and deleted allowing the recursive delete function to re-examine the parent key. The process is repeated for each subkey branch until no subkey branches remain. Then the particular key specified may itself be deleted.

A point to remember when enumerating and deleting subkeys is to always enumerate subkey index zero (that is, DWORD iSubkey = 0). Because keys are reindexed after each key is deleted, the use of a non-zero subkey index would result in keys not being deleted. This in turn would result in the failure of the RegDeleteKey function when an attempt is made to delete the subkey's parent key.

Partial Deletions

Failure to fully delete the particular key specified can be the result of

'partial deletions' (the failure to delete all available subkeys). Although partial deletions can result from several situations, one possible cause is individual key protection.

To protect against partial deletions caused by protected keys, you should test each individual key to ensure that it is not protected from deletion. To test if the current user has deletion rights on all the keys to be deleted, you must traverse, enumerate, and open all subkeys with DELETE privilege requested:

```
   RegOpenKeyEx(
      hStartKey,pKeyName, 0,
      KEY_ENUMERATE_SUB_KEYS | DELETE,
      &hKey ))
```

If, however, between the time of the delete privilege test and the actual attempt to delete, the key protection is altered, the recursive delete function will still fail.

To truly protect the registry against partial deletion, you need to follow a two-step process. First, prior to the deletion attempt, save the initial state of the registry path to be deleted. Then, to recover from a partial deletion, you could restore the registry to its former state using the information already saved. If partial deletions are acceptable, however, failure to delete a key could trigger the recursive delete function to fail or the key to be skipped.

Sample Code

```
// The sample code makes no attempt to check or recover from partial
// deletions.
//
// A registry key that is opened by an application can be deleted
// without error by another application in both Windows 95 and
// Windows NT. This is by design.

DWORD RegDeleteKeyNT(HKEY hStartKey , LPTSTR pKeyName )
{
   DWORD   dwRtn, dwSubKeyLength;
   LPTSTR  pSubKey = NULL;
   TCHAR   szSubKey[MAX_KEY_LENGTH]; // (256) this should be dynamic.
   HKEY    hKey;

   // do not allow NULL or empty key name
   if ( pKeyName &&  lstrlen(pKeyName))
   {
      if( (dwRtn=RegOpenKeyEx(hStartKey,pKeyName,
         0, KEY_ENUMERATE_SUB_KEYS | DELETE, &hKey )) == ERROR_SUCCESS)
      {
         while (dwRtn == ERROR_SUCCESS )
         {
            dwSubKeyLength = MAX_KEY_LENGTH;
            dwRtn=RegEnumKeyEx(
                        hKey,
```

```
                           0,       // always index zero
                           szSubKey,
                           &dwSubKeyLength,
                           NULL,
                           NULL,
                           NULL,
                           NULL
                         );

            if(dwRtn == ERROR_NO_MORE_ITEMS)
            {
               dwRtn = RegDeleteKey(hStartKey, pKeyName);
               break;
            }
            else if(dwRtn == ERROR_SUCCESS)
               dwRtn=RegDeleteKeyNT(hKey, szSubKey);
         }
         RegCloseKey(hKey);
         // Do not save return code because error
         // has already occurred
      }
   }
   else
      dwRtn = ERROR_BADKEY;

   return dwRtn;
}
```

Additional reference words:
KBCategory: kbusage kbprg kbhowto kbcode
KBSubcategory: BseMisc

How to Implement Context-Sensitive Help in Windows 95 Dialogs

PSS ID Number: Q125670

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

In Windows versions 3.x, applications implement context-sensitive help for dialog boxes by installing either a message filter hook, or a task-specific keyboard hook that monitors the WM_KEYDOWN message and responds to F1 key presses.

Windows 95 makes it easier because it provides a new WM_HELP message that gets sent each time the user presses the F1 key, giving the application a chance to bring up help information on the control that has the keyboard focus or on the dialog box itself. This new WM_HELP message is not limited to dialog boxes alone, as it gets sent to any window that has keyboard focus or to the currently active window.

MORE INFORMATION

Windows 95 also provides a new dialog style, DS_CONTEXTHELP that adds a question mark button to the dialog box's caption bar. This button, when clicked, changes the cursor to a question mark with a pointer. When the user clicks any control in the dialog box, Windows 95 sends a WM_HELP message for that control. The dialog procedure should process the WM_HELP message as follows:

```
   // Define an array of dword pairs,
   // where the first of each pair is the control ID,
   // and the second is the context ID for a help topic,
   // which is used in the help file.
   static const DWORD aMenuHelpIDs[] =
   {
      edt1, IDH_EDT1 ,
      lst1, IDH_LST1 ,
      lst2, IDH_LST2 ,
      0,    0
   };

   case WM_HELP:
   {
      LPHELPINFO lphi;

      lphi = (LPHELPINFO)lparam;
      if (lphi->iContextType == HELPINFO_WINDOW)   // must be for a control
      {
```

```
         WinHelp (lphi->hItemHandle,
                  "GEN32.HLP",
                  HELP_WM_HELP,
                  (DWORD)(LPVOID)aMenuHelpIDs);
      }
      return TRUE;
   }
```

Calling WinHelp() with the HELP_WM_HELP parameter as demonstrated above displays the help topic in a pop-up window.

In addition to the WM_HELP message, Windows 95 provides a new WM_CONTEXTMENU message that gets sent each time the user right-clicks a window. Typically this message is processed by displaying a context menu using the TrackPopupMenu() function. However, this message can be processed to bring up help information by calling the WinHelp() function to display the help topic in a pop-up window, as in this example:

```
   case WM_CONTEXTMENU:
   {
      WinHelp ((HWND)wparam,
               "GEN32.HLP",
                HELP_CONTEXTMENU,
               (DWORD)(LPVOID)aSampleMenuHelpIDs);
      return TRUE;
   }
```

NOTE: Look at the third parameter (HELP_CONTEXTMENU) passed to WinHelp() this time. This causes a pop-up menu to come up that displays "What's This?" text. It then displays the help topic in a pop-up window when the menu item is selected.

Additional reference words: 4.00
KBCategory: kbui kbcode
KBSubcategory: UsrDlgs

How to Increase Windows NT System and Desktop Heap Sizes

PSS ID Number: Q125752

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SUMMARY

Sometimes, it may be necessary to increase the amount of memory that Windows NT will make available for the system and desktop heaps. This can be accomplished by editing an entry in the registration database. System heap items are things like desktops and one-time-allocated items like system metrics. The items that come out of the desktop heap are items such as windows, menus, hook structures, queues, and some thread information.

MORE INFORMATION

The entry to be edited is under:

```
HKEY_LOCAL_MACHINE\
   System\
   CurrentControlSet\
   Control\
   Session Manager\
   SubSystems\
   Windows
```

Under this entry, you will find a string similar to the following (the slash (/) is a line continuation charater):

```
%SystemRoot%\system32\csrss.exe /
   ObjectDirectory=\Windows /
   SharedSection=1024,512 /
   Windows=On /
   SubSystemType=Windows /
   ServerDll=basesrv,1 /
   ServerDll=winsrv:GdiServerDllInitialization,4 /
   ServerDll=winsrv:UserServerDllInitialization,3 /
   ServerDll=winsrv:ConServerDllInitialization,2 /
   ProfileControl=Off /
   MaxRequestThreads=16
```

By changing the SharedSection values, you can affect the heap sizes. The first number (1024 as shown above) is the maximum size of the system wide heap in kilobytes. The second number (512 as shown above) is the maximum size of the per desktop heap in kilobytes. A desktop value of 512K can support approximately 2,500 windows.

The memory you allocate needs to be backed up by paging space. It should not have much effect on performance if you create the same number of items with different heap sizes. The main effect is overhead in heap management and initialization.

Additional reference words: 3.50
KBCategory: kbui
KBSubcategory: UsrWndw

How to Install TAPI Service Providers with TELEPHON.CPL

PSS ID Number: Q132190

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0
- Microsoft Windows Telephony Software Development Kit (TAPI SDK) version 1.0

SUMMARY

TAPI Service Providers (TSPs) must have version information to be displayed in the 'Add Driver' dialog in the Telephony Control Panel Applet (CPL).

MORE INFORMATION

The Telephony CPL reads the version information from all TSPs in the Windows System directory to get the strings displayed in the Add Driver dialog box. If any particular TSP doesn't include the necessary version information, it isn't included as an installable Telephony Driver. The ATSP sample demonstrates how to include the version information.

This is not an issue with Windows 95. Under Windows 95, Telephony CPL uses the filename of the TSP if the TSP doesn't include version information. However, Microsoft recommends that version information still be included so that the user can use a friendly name rather than the filename when installing the TSP.

Additional reference words: 1.00 1.30 1.40 4.00 95
KBCategory: kbprg
KBSubcategory: TAPI

How to Keep a Window Iconic

PSS ID Number: Q66244

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Normally, when an application's main window is being represented by an icon ("iconic"), you can restore it to an open window by double-clicking the icon or by choosing the Restore option from the System menu.

Opening the window can be prevented by placing code into the application that processes the WM_QUERYOPEN message by returning FALSE.

If it is necessary to perform processing before the iconic window is opened, the processing should be done in response to the WM_QUERYOPEN message. After processing is complete the program can return TRUE and the window will be opened.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

How to Keep an MDI Window Always on Top

PSS ID Number: Q108315

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

When creating a multiple document interface (MDI) window, there are no styles available to have the new window stay on top of the other MDI windows. Alternatively, two methods are available to achieve this functionality:

- Process the WM_WINDOWPOSCHANGED message and call SetWindowPos() to change the Z-order of the window.

- Install a timer for the MDI windows and reset the Z-order of the window when processing the WM_TIMER message.

MORE INFORMATION
================

MDICREATESTRUCT has the field "style", which can be set with the styles for the new MDI window. Extended styles, such as WS_EX_TOPMOST, are not available in MDI windows. This field of MDICREATESTRUCT is passed to CreateWindowEx() in the dwStyle parameter. The dwExStyle field is set to 0L. The two methods shown below cannot be used at the same time in the same application.

Method 1: Process the WM_WINDOWPOSCHANGED message and call SetWindowPos() to change the Z-order of the window.

Sample Code

```
LRESULT CALLBACK MdiWndProc (HWND hWnd, UINT message, WPARAM wParam,
                    LPARAM lParam)
{
    static HWND hWndAlwaysOnTop = 0;
    switch (message)
     {
     case WM_CREATE :
         if (!hWndAlwaysOnTop)
          {
          SetWindowText (hWnd, "Always On Top Window");
          hWndAlwaysOnTop = hWnd;
```

```
          }
            break;
     case WM_WINDOWPOSCHANGED :
         if (hWndAlwaysOnTop)
          {
          WINDOWPOS FAR* pWP = (WINDOWPOS FAR*)lParam;
          if (pWP->hwnd != hWndAlwaysOnTop)
              SetWindowPos (hWndAlwaysOnTop, HWND_TOP, 0, 0, 0, 0,
                      SWP_NOACTIVATE | SWP_NOMOVE | SWP_NOSIZE);
          }
         break;
     //
     // Other Messages to process here.
     //
     case WM_CLOSE :
         if (hWndAlwaysOnTop == hWnd)
          hWndAlwaysOnTop = NULL;
     default :
         return DefMDIChildProc (hWnd, message, wParam, lParam);
     }
    return 0L;
}
```

Method 2: Install a timer for the MDI windows and reset the Z-order of the window when processing the WM_TIMER message.

Sample Code

```
LRESULT CALLBACK MdiWndProc (HWND hWnd, UINT message, WPARAM wParam,
                   LPARAM lParam)
{
    static HWND hWndAlwaysOnTop = 0;
    switch (message)
     {
     case WM_CREATE :
         SetTimer (hWnd, 1, 200, NULL);
         if (!hWndAlwaysOnTop)
          {
          SetWindowText (hWnd, "Always On Top Window");
          hWndAlwaysOnTop = hWnd;
          }
            break;
     case WM_TIMER :
         if (hWndAlwaysOnTop)
          {
          SetWindowPos (hWndAlwaysOnTop, HWND_TOP, 0, 0, 0, 0,
                     SWP_NOACTIVATE | SWP_NOMOVE | SWP_NOSIZE);
          }
         break;
     case WM_DESTROY:
         KillTimer (hWnd, 1) ;
         break;
     //
     // Other Messages to process here.
```

```
     //
     case WM_CLOSE :
         if (hWndAlwaysOnTop == hWnd)
          hWndAlwaysOnTop = NULL;
     default :
         return DefMDIChildProc (hWnd, message, wParam, lParam);
     }
    return 0L;
}
```

For additional information on changing the Z-order of child pop-up windows, please see the following article(s) in the Microsoft Knowledge Base:

```
   ARTICLE-ID: Q66943
   TITLE     : Determining the Topmost Pop-Up Window
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

How to License the MS LineDraw TrueType Font

PSS ID Number: Q150641

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

- Microsoft Win32 Application Programming Interface included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SUMMARY

Some Microsoft applications, including Microsoft Word, are shipped with a TrueType font named MS LineDraw. This font is located in the "linedraw.ttf" font file and contains glyphs for the line drawing characters of the extended ASCII character set. Many earlier MS-DOS applications use the extended ASCII MS LineDraw characters to produce text-based graphics.

Windows application developers have expressed interest in the MS LineDraw font as a means of maintaining some form of backward compatibility with their previous MS-DOS-based applications. Developers can use the MS LineDraw font in their applications by licensing it from Monotype. To license MS LineDraw, contact Wade Farrell at Monotype:

Telephone: 1-800-MONOTYPE, (312)855-1440

- or -

E-mail: oemsales@monotypeusa.com

MORE INFORMATION

An alternative to licensing MS LineDraw is to use the GDI functions of the Windows SDK to directly draw any graphics required by the application. By using GDI, developers eliminate the potentially troublesome task of aligning glyphs on various and differing devices. Backward compatibility is achieved by converting the data to the representative characters from the extended ASCII character set when it is interchanged with the MS-DOS-based applications.

Additional reference words: 3.10 3.11 3.50 3.51 4.00
KBCategory: kbgraphic kbother kbref kbhowto
KBSubcategory: GdiFnt GdiTt

How to Limit Virtual Memory for a Win32s Application

PSS ID Number: Q147434

The information in this article applies to:

- Microsoft Win32s version 1.3c

SUMMARY

=======

It is possible to limit the amount of virtual memory that a 32-bit application running under Win32s version 1.3c can use. This could be a potential requirement for some 32-bit applications that do not want to swap data out to the pagefile at all.

MORE INFORMATION:

================

In your Win32s application, you can call VirtualAlloc() to allocate all the required memory at one time and then use only this memory.

To do this, you need to adjust the working set size of your 32-bit process. Win32s partially supports the Win32 functions GetProcessWorkingSet() and SetProcessWorkingSet(). Although the hProcess and dwMinimumWorkingSetSize parameters are ignored, you can call SetProcessWorkingSet(), and set the dwMaximumWorkingSetSize to the total number of pages that your 32-bit application needs to lock in memory. This number is system wide in the Windows 3.x environment; it is not a per-process number as in Windows NT.

Next, you need to call VirtualLock() to lock the allocated pages. This way the memory will not be swapped out. Note that VirtualLock() is supported in Win32s even though the Win32 documentation says otherwise. It is also important to note that unlike in Win32, the dwMaximumWorkingSetSize parameter of SetProcessWorkingSet() function does not limit the working set size of Win32s applications. It only sets the limit on how many pages can be locked in memory (by calling VirtualLock()).

Additional reference words: 1.30c kbinf noswap pagefile virtual memory
KBCategory: kbprg kbdocerr kbhowto
KBSubcategory: w32s

How to Look Up a User's Full Name

PSS ID Number: Q119670

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

SUMMARY

Windows NT workstations can be organized into a domain, which is a collection of computers on a Windows NT Advanced Server network. The domain administrator maintains centralized user and group account information.

MORE INFORMATION

To find the full name of a user if you have the user name and domain name:

1. Convert the user name and domain name to Unicode, if they are not already Unicode strings. This is a requirement of the ported LAN Manager APIs that are used in the following steps.

2. Look up the name of the domain controller (DC) for the domain name by calling NetServerEnum().

3. Look up the user name by calling NetUserGetInfo()

4. Convert the full user name to ANSI, unless the program is expecting to work with Unicode strings.

The sample code below is a function that takes a user name and a domain name as the first two arguments and returns the user's full name in the third argument.

For information on how to get the current user and domain, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q111544
TITLE : Looking Up the Current User and Domain

Sample Code

```
#include <windows.h>
#include <lm.h>
#include <stdio.h>

/*****************************************************************\
* Function: GetFullName( char *UserName, char *Domain, char *dest ); *
*                                                                 *
* Parameters:                                                     *
```

```
*    UserName: the user name                                          *
*    Domain  : the domain to which the user belongs                   *
*    dest    : receives the user's full name                          *
*                                                                     *
\*********************************************************************/

BOOL GetFullName(char *UserName, char *Domain, char *dest)
{
   WCHAR  wszUserName[256];           // Unicode user name
   WCHAR  wszDomain[256];
   LPBYTE ComputerName;

   struct _SERVER_INFO_100 *si100;   // Server structure
   struct _USER_INFO_2 *ui;          // User structure

   // Convert ASCII user name and domain to Unicode.

   MultiByteToWideChar( CP_ACP, 0, UserName,
      strlen(UserName)+1, wszUserName, sizeof(wszUserName) );
   MultiByteToWideChar( CP_ACP, 0, Domain,
      strlen(Domain)+1, wszDomain, sizeof(wszDomain) );

   // Get the computer name of a DC for the specified domain.

   NetGetDCName( NULL, wszDomain, &ComputerName );

   // Look up the user on the DC.

   if(NetUserGetInfo( (LPWSTR) ComputerName,
      (LPWSTR) &wszUserName, 2, (LPBYTE *) &ui))
   {
      printf( "Error getting user information.\n" );
      return( FALSE );
   }

   // Convert the Unicode full name to ASCII.

   WideCharToMultiByte( CP_ACP, 0, ui->usri2_full_name,
      -1, dest, 256, NULL, NULL );

   return( TRUE );
}
```

Additional reference words: 3.10 3.50
KBCategory: kbnetwork kbnetwork
KBSubcategory: NtwkLmapi

How to Make an Application Display Real Units of Measurement

PSS ID Number: Q127152

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

Sometimes you need an application to display things in terms of a real unit of measurement such as an inch or millimeter. When dealing with a printer, resolution is usually given in dots per inch (DPI), which makes it easy to convert pixels to real inches. However, on a video display, resolution is given only in pixels. A given video mode will be some X pixels wide with no information as to the real dimensions of the display area.

Because there is no way to programmatically determine the real dimensions of the viewable area on a video display, it is impossible for a program to determine real output dimensions. Two manual methods for determining real output dimensions are given in this article.

MORE INFORMATION
================

When output is destined for a printer, the application can call GetDeviceCaps() using LOGPIXELSX and LOGPIXELSY to determine dots per real inch. However, for a video display, LOGPIXELSX and LOGPIXELSY are defined by the video driver and may vary wildly. These numbers define a logical inch, which is almost never equal to a real inch.

Applications that need to output real sizes to the video display can use one of the following two methods for determining output size:

1. The application can ask the user what size monitor is attached. Using this value, an application can approximate the actual viewable area, and given the resolution of the output (GetDeviceCaps, HORZRES, VERTRES), the application can approximate real inches. This solution gives only an approximation of a real inch. Several factors can introduce errors into this approximation including the size adjustments on digital monitors.

2. The application can ask the user to hold a measuring device to the screen and measure a given line. This is the only way to guarantee that output on a video display is exactly the expected size, and recalibration would be necessary after any adjustment to the monitor.

Additional reference words: 3.10 3.50 4.00 95

KBCategory: kbgraphic
KBSubcategory: GdiDisplay

How to Manage Computer Accounts Programmatically in Windows NT

PSS ID Number: Q136867

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SUMMARY

=======

The ability to manage Windows NT computer accounts can be accomplished by using the Windows NT server manager utility. The server manager (srvmgr) can be used to create and delete various computer accounts, including the following account types:

- Windows NT Workstation
- Windows NT Server (non domain controller)
- Windows NT Backup domain controller

Furthermore, Windows NT user manager for domains (usrmgr) provides the ability to manage trust relationships with domains. The management of interdomain trust accounts "permitted to trust this domain" will be addressed by this article.

The list of trusted domains can be managed programmatically with the Windows NT LSA (local security authority) API. This procedure will not be discussed in this article.

This article describes how to manage Windows NT computer accounts programmatically by using Win32 API calls from Windows NT. It is assumed that you already have an understanding of the various computer account types in Windows NT.

MORE INFORMATION

================

Considerations for Managing Computer Accounts

- Computer account management should take place on the primary domain controller for the target domain.

- The computer account name should be all uppercase for consistency with Windows NT account management utilities.

- A computer account name always has a trailing dollar sign ($). Any APIs used to manage computer accounts must build the computer name such that the last character of the computer account name is a dollar sign ($). For interdomain trust, the account name is TrustingDomainName$.

- The maximum computer name length is MAX_COMPUTERNAME_LENGTH (15). This length does not include the trailing dollar sign ($).

- The password for a new computer account should be the lowercase representation of the computer account name, without the trailing dollar sign ($). For interdomain trust, the password can be an arbitrary value that matches the value specified on the trust side of the relationship.

- The maximum password length is LM20_PWLEN (14). The password should be truncated to this length if the computer account name exceeds this length.

- The password provided at computer-account-creation time is valid only until the computer account becomes active on the domain. A new pasword is established during trust relationship activation.

- The user that calls the account management functions must have Administrator privilege on the target computer. In the case of existing computer accounts, the creator of the account can manage the account, regardless of administrative membership.

- The SeMachineAccountPrivilege can be granted on the target computer to give specified users the ability to create computer accounts. This gives non-administrators the ability to create computer accounts. The caller needs to enable this privilege prior to adding the computer account.

- The Windows NT Lan Manager APIs are currently implemented as Unicode only. The caller must insure that strings passed to these functions are in Unicode form.

Machine account types are defined by the following flags:

```
UF_SERVER_TRUST_ACCOUNT (Backup domain controller)
UF_WORKSTATION_TRUST_ACCOUNT (Workstation and server)
UF_INTERDOMAIN_TRUST_ACCOUNT (Interdomain trust account)
```

The following list of APIs can be used to manage computer accounts in the target domain:

- NetGetDCName can be used to obtain the computer name of the primary domain controller.

- NetUserAdd can be used for creating a new computer account.

- NetUserDel can be used to delete an existing computer account.

- NetUserSetInfo can be used to modify the password of an existing computer account. This is useful for resetting a computer account to a known state.

- NetUserEnum can be used to enumerate existing computer accounts. This API can return a list of accounts based on account type through the use of the filter parameter.

Sample Code

```
// works compiled ansi or unicode
```

```
#define UNICODE
#define _UNICODE

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

/*++

The following sample code adds the specified workstation computer account
to the specified domain. If no domain is specified, the computer account is
created on the local computer.

If the computer account creation fails with GetLastError ==
ERROR_ACCESS_DENIED, the sample attempts to enable the
SeMachineAccountPrivilege for the caller. If the privilege is enabled
successfully, the computer account add operation is re-tried.

The following import libraries are required:

   netapi32.lib
   advapi32.lib

--*/

#include <windows.h>
#include <stdio.h>
#include <lm.h>

BOOL
AddMachineAccount(
    LPWSTR wTargetComputer,
    LPWSTR MachineAccount,
    DWORD AccountType
    );

BOOL SetCurrentPrivilege(
    LPWSTR TargetComputer,  // target of privilege operation
    LPCWSTR Privilege,      // Privilege to enable/disable
    BOOL bEnablePrivilege   // to enable or disable privilege
    );

int wmain(int argc, wchar_t *argv[])
{
    LPWSTR wMachineAccount;
    LPWSTR wPrimaryDC;
    LPWSTR wMachineAccountPrivilege = L"SeMachineAccountPrivilege";
    DWORD dwError;
    BOOL bSuccess;

    if (argc < 2)
    {
        fprintf(stderr, "Usage: %ls <machineaccountname> [domain]\n",
            argv[0]);
        return RTN_USAGE;
```

```
}

wMachineAccount = argv[1];

//
// default will operate on local machine.  Non-NULL wPrimaryDC will
// cause buffer to be freed
//
wPrimaryDC = NULL;

//
// if a domain name was specified, fetch the computer name of the
// primary domain controller
//
if (argc == 3) {

    dwError = NetGetDCName(NULL, argv[2], (LPBYTE *)&wPrimaryDC);

    if(dwError != NO_ERROR) {
        fprintf(stderr,"NetGetDCName error! (rc=%lu)\n", dwError);
        return RTN_ERROR;
    }
}

bSuccess=AddMachineAccount(
    wPrimaryDC,                 // primary DC computer name
    wMachineAccount,            // computer account name
    UF_WORKSTATION_TRUST_ACCOUNT // computer account type
    );

if(!bSuccess && GetLastError() == ERROR_ACCESS_DENIED ) {

    //
    // try to enable the SeMachineAccountPrivilege
    //
    if(SetCurrentPrivilege(
        wPrimaryDC, wMachineAccountPrivilege, TRUE )) {

        //
        // enabled the privilege.  retry the add operation
        //
        bSuccess=AddMachineAccount(
            wPrimaryDC,
            wMachineAccount,
            UF_WORKSTATION_TRUST_ACCOUNT
            );

        //
        // disable the privilege
        //
        SetCurrentPrivilege(
            wPrimaryDC, wMachineAccountPrivilege, FALSE);
    }
}
```

```
    //
    // free the buffer allocated for the PDC computer name
    //
    if(wPrimaryDC) NetApiBufferFree(wPrimaryDC);

    if(!bSuccess)
    {
        fprintf(stderr,"AddMachineAccount error! (rc=%lu)\n",
            GetLastError());
        return RTN_ERROR;
    }

    return RTN_OK;
}

BOOL
AddMachineAccount(
    LPWSTR wTargetComputer,
    LPWSTR MachineAccount,
    DWORD AccountType
    )
{
    LPWSTR wAccount;
    LPWSTR wPassword;
    USER_INFO_1 ui;
    DWORD cbAccount;
    DWORD cbLength;
    DWORD dwError;

    //
    // ensure a valid computer account type was passed
    // TODO SetLastError
    //
    if (AccountType != UF_WORKSTATION_TRUST_ACCOUNT &&
        AccountType != UF_SERVER_TRUST_ACCOUNT &&
        AccountType != UF_INTERDOMAIN_TRUST_ACCOUNT
        ) {
        SetLastError(ERROR_INVALID_PARAMETER);
        return FALSE;
    }

    //
    // obtain number of chars in computer account name
    //
    cbLength = cbAccount = lstrlenW(MachineAccount);

    //
    // ensure computer name doesn't exceed maximum length
    //
    if(cbLength > MAX_COMPUTERNAME_LENGTH) {
        SetLastError(ERROR_INVALID_ACCOUNT_NAME);
        return FALSE;
    }

    //
```

```
// allocate storage to contain Unicode representation of
// computer account name + trailing $ + NULL
//
wAccount=(LPWSTR)HeapAlloc(GetProcessHeap(), 0,
    (cbAccount + 1 + 1) * sizeof(WCHAR)  // Account + '$' + NULL
    );

if(wAccount == NULL) return FALSE;

//
// password is the computer account name converted to lowercase
// you will convert the passed MachineAccount in place
//
wPassword = MachineAccount;

//
// copy MachineAccount to the wAccount buffer allocated while
// converting computer account name to uppercase.
// convert password (inplace) to lowercase
//
while(cbAccount--) {
    wAccount[cbAccount] = towupper( MachineAccount[cbAccount] );
    wPassword[cbAccount] = towlower( wPassword[cbAccount] );
}

//
// computer account names have a trailing Unicode '$'
//
wAccount[cbLength] = L'$';
wAccount[cbLength + 1] = L'\0'; // terminate the string

//
// if the password is greater than the max allowed, truncate
//
if(cbLength > LM20_PWLEN) wPassword[LM20_PWLEN] = L'\0';

//
// initialize USER_INFO_x structure
//
ZeroMemory(&ui, sizeof(ui));

ui.usri1_name = wAccount;
ui.usri1_password = wPassword;

ui.usri1_flags = AccountType | UF_SCRIPT;
ui.usri1_priv = USER_PRIV_USER;

dwError=NetUserAdd(
            wTargetComputer,    // target computer name
            1,                  // info level
            (LPBYTE) &ui,       // buffer
            NULL
            );

//
```

```
    // free allocated memory
    //
    if(wAccount) HeapFree(GetProcessHeap(), 0, wAccount);

    //
    // indicate whether it was successful
    //
    if(dwError == NO_ERROR)
        return TRUE;
    else {
        SetLastError(dwError);
        return FALSE;
    }
}

BOOL SetCurrentPrivilege(
    LPWSTR TargetComputer,  // target of privilege operation
    LPCWSTR Privilege,      // Privilege to enable/disable
    BOOL bEnablePrivilege   // to enable or disable privilege
    )
{
    HANDLE hToken;
    TOKEN_PRIVILEGES tp;
    LUID luid;
    TOKEN_PRIVILEGES tpPrevious;
    DWORD cbPrevious=sizeof(TOKEN_PRIVILEGES);
    BOOL bSuccess=FALSE;

    if(!LookupPrivilegeValueW(TargetComputer, Privilege, &luid))
            return FALSE;

    if(!OpenProcessToken(
            GetCurrentProcess(),
            TOKEN_QUERY | TOKEN_ADJUST_PRIVILEGES,
            &hToken
            )) return FALSE;

    //
    // first pass.  get current privilege setting
    //
    tp.PrivilegeCount           = 1;
    tp.Privileges[0].Luid       = luid;
    tp.Privileges[0].Attributes = 0;

    AdjustTokenPrivileges(
            hToken,
            FALSE,
            &tp,
            sizeof(TOKEN_PRIVILEGES),
            &tpPrevious,
            &cbPrevious
            );

    if(GetLastError() == ERROR_SUCCESS) {
        //
```

```
        // second pass.  set privilege based on previous setting
        //
        tpPrevious.PrivilegeCount     = 1;
        tpPrevious.Privileges[0].Luid = luid;

        if(bEnablePrivilege) {
            tpPrevious.Privileges[0].Attributes |=
                (SE_PRIVILEGE_ENABLED);
        }
        else {
            tpPrevious.Privileges[0].Attributes ^=
                (SE_PRIVILEGE_ENABLED &
                tpPrevious.Privileges[0].Attributes);
        }

        AdjustTokenPrivileges(
                hToken,
                FALSE,
                &tpPrevious,
                cbPrevious,
                NULL,
                NULL
                );

        if (GetLastError() == ERROR_SUCCESS) bSuccess=TRUE;
    }

    CloseHandle(hToken);

    return bSuccess;
}
```

Additional reference words: computer account server workstation trust
KBCategory: kbnetwork kbcode
KBSubcategory: BseSecurity NtWkLmapi CodeSam

How to Manage Trusted Domains Programmatically in Windows NT

PSS ID Number: Q145697

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.51

SUMMARY
=======

The ability to manage domain trust in a Windows NT environment can be accomplished by using the "Windows NT User Manager for Domains" utility (usrmgr). By choosing Trust Relationships on the Policies menu, a user can manage both sides ("Trusted Domains" and "Permitted to trust this domain") of the domain trust relationship on a domain.

This article illustrates the programmatic manipulation of the "Trusted Domains" side of the domain trust relationship, at the domain controller level. This article does not address management of "Trusted Domains" at the workstation level, which is used for managing workstation domain membership.

To understand this article, you need to understand how the domain trust relationship affects the behavior of Windows NT. It is assumed that you have this understanding.

MORE INFORMATION
================

The management of interdomain trust accounts ("permitted to trust this domain") is be addressed by this article. For information about managing the "permitted to trust this domain" side of the domain trust relationship, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q136867
TITLE   :How to Manage Computer Accounts Programmatically in Windows NT

You can manage trusted domains and associated passwords programmatically by using the Windows NT LSA (local security authority) API. Use the Windows NT LAN Manager API to obtain information necessary to manage trusted domains.

Considerations for Managing Trusted Domains

- Trusted domain updates must take place on the primary domain controller for the target domain.

- The trusted domain name should be all uppercase for consistency with Windows NT account management utilities.

- The maximum domain name length is MAX_COMPUTERNAME_LENGTH (15).

- The maximum trusted domain password length is LM20_PWLEN (14).

- The password provided at trust-creation time is valid only until the trust has become active on the domain. A new password is established during trust relationship activation.

- The user who calls the LSA trust management functions must have Administrator privilege on the target domain.

- The Windows NT LSA API and Windows NT LAN Manager API are currently implemented as Unicode only. The caller must ensure that strings passed to these functions are in Unicode form.

- LsaSetTrustedDomainInformation does not verify the validity or existence of the supplied domain Sid. The caller of these API is responsible for ensuring that a valid Sid is provided to the API. You can use NetUserModalsGet at info level 2 against the domain controller for the trusted domain for this purpose.

- Do not set TrustedPosixOffsetInformation info level with LsaSetTrustedDomainInformation(). This information is managed by the LSA, so it should not be set.

- Do not set a domain to trust itself; the trusted domain name should not match the domain name where the policy update takes place.

List of APIs That Can Be Used to Manage Trusted Domains

NetGetDCName can be used to obtain the computer name of the primary domain controller associated with a domain name.

NetUserModalsGet at info level 2 can be used to obtain the Sid representing the trusted domain, in addition to the domain name of the specified computer. The target machine is a domain controller for the trusted domain. The resultant domain Sid is supplied to LSA API calls that require the Domain Sid of the trusted domain to manage.

NetServerGetInfo at info level 101 can be used to determine if the target of the policy object update is a domain controller or a workstation. This article does not address the workstation case, so this determination is necessary.

NetApiBufferFree is used to free buffers allocated by Windows NT LAN Manager APIs that return information.

LsaOpenPolicy is used to open the policy object on the computer where the trusted domain update is taking place. The target computer is a primary domain controller.

LsaDeleteTrustedDomain is used to delete a trusted domain.

LsaQueryTrustedDomainInfo at info level TrustedPasswordInformation is used to determine if the trusted domain object already exists, and if so, what the current Password is.

LsaSetTrustedDomainInformation at info level TrustedDomainNameInformation is used to create the trusted domain object if the object does not yet exist.

LsaSetTrustedDomainInformation at info level TrustedPasswordInformation is used to set the new password associated with the trusted domain object. If the specified trust existed after the call to LsaQueryTrustedDomainInfo, set the OldPassword on the trusted domain object to the previous password obtained from the Query operation.

LsaFreeMemory should be used if memory was allocated by the LSA for the caller. An example of this is data provided by LsaQueryTrustedDomainInfo.

LsaCloseHandle is used to close an open policy handle when it is no longer needed.

Steps to Verify That the Trust Exists

Prior to creating a new trusted domain, it may be useful to verify that the "permitted to trust this domain" side of the trust relationship exists and is consistent with the new trusted domain. If the trust does not exist or is inconsistent, the user could be alerted to take action in order to complete the trust relationship.

The trust can be verified by attempting to establish a connection to the trusted domain and specifying a username, domain name, and password corresponding to the trust account and trusted domain password.
The following steps should be used to verify the trust:

1. Obtain the domain controller computer name for the trusted domain by calling NetGetDCName(). This domain name is also used as the domain name associated with the trust account.

2. Build a UNC path containing the trusted domain computer name and the Interprocess communication share. An example UNC path is:

   \\WINBASE\IPC$.

3. Build a user name consisting of the domain name, which is "permitted to trust," with a "$" appended. An example user name is:

   MYDOMAIN$

4. Call NetUseAdd() at info level 2 to specify the domain name, UNC path, user name, and password associated with the trust.

If the trust is successfully verified, the connection attempt fails with ERROR_NOLOGON_INTERDOMAIN_TRUST_ACCOUNT. In this case, the supplied credentials were validated, but the account type does not allow a network style logon.

If NetUseAdd succeeds with NERR_Success, the specified trust account does not exist, because the specified account does not exist on the target

computer, and default credentials were used to establish the connection.
Use NetUseDel to delete the connection.

If NetUseAdd fails with ERROR_LOGON_FAILURE, the supplied password does not match the password on the trust account.

If NetUseAdd fails with ERROR_SESSION_CREDENTIAL_CONFLICT, a connection already exists to the specified domain controller, which uses credentials that differ from the credentials supplied with NetUseAdd. The existing connection must be deleted to verify the trust.

NOTE: This method of trust verification is advisory and may be specific to versions of Windows NT versions 3.1 through 4.0.

Sample Code

```
/*++

 This sample illustrates how to manage Windows NT trusted domains at the
 domain controller level.

 The first command line argument indicates the name of the new or existing
 trusted domain to create or modify.

 The second command line argument indicates the new password for the
 trusted domain specified in the first argument.

 The optional third argument indicates the domain name that is the target
 of the trusted domain update operation. If this argument is not specified,
 the update will occur on the local domain. Note that this sample will not
 allow a trusted domain update to occur on a non-domain controller.

 This sample requires header files found on the Windows NT 3.51 SDK CD-ROM
 compact disc in the \Mstools\Security directory.

 This sample works correctly compiled ANSI or Unicode. Note that LAN
 Manager NetXxx API are Unicode only, and Windows NT LSA API are Unicode
 only.

 The following import libraries are required:

 Netapi32.lib
 Advapi32.lib

--*/

#ifndef UNICODE
#define UNICODE
#define _UNICODE
#endif

#include <windows.h>
#include <lm.h>         // for NetXxx API
#include "ntsecapi.h"   // \Mstools\Security\Ntsecapi.h
```

```
#include <stdio.h>

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

//
// if you have the DDK, include Ntstatus.h
//
#ifndef STATUS_SUCCESS
#define STATUS_SUCCESS                  ((NTSTATUS)0x00000000L)
#define STATUS_OBJECT_NAME_NOT_FOUND    ((NTSTATUS)0xC0000034L)
#define STATUS_INVALID_SID              ((NTSTATUS)0xC0000078L)
#endif

void
InitLsaString(
    PLSA_UNICODE_STRING LsaString,
    LPWSTR String
    );

NTSTATUS
OpenPolicy(
    LPWSTR ServerName,
    DWORD DesiredAccess,
    PLSA_HANDLE PolicyHandle
    );

BOOL
GetDomainSid(
    LPWSTR DomainName,  // domain name to acquire Sid of
    PSID *pDomainSid    // points to allocated Sid on success
    );

NTSTATUS
SetTrustedDomainInfo(
    LSA_HANDLE PolicyHandle,
    PSID DomainSid,             // Sid of domain to manipulate
    LPWSTR TrustedDomainName,   // trusted domain name to add/update
    LPWSTR Password             // new trust password for trusted domain
    );

BOOL
VerifyTrustRelationship(
    LPWSTR TargetDomainName,        // domain name to verify trust at
    LPWSTR TrustAccountToVerify,    // trusted domain name to verify
    LPWSTR Password,                // password associated with trust
    LPBOOL bTrustVerified           // indicates if trust was verified
    );

void
DisplayNtStatus(
    LPSTR szAPI,    // ANSI string containing API name
    NTSTATUS Status
```

```
    );

void
DisplayError(
    LPSTR szAPI,    // pointer to failed API name
    DWORD dwLastError
    );

//
// Unicode entry point and argv
//
int
__cdecl
wmain(
    int argc,
    wchar_t *argv[]
    )
{
    BOOL bAllocTargetDomain;    // did you allocate memory for target?
    LSA_HANDLE PolicyHandle;
    LPWSTR TargetDomainName;    // target domain of policy update
    LPWSTR DomainController;    // computer name where policy update
    PSID DomainSid; // allocated Sid representing domain to trust
    LPWSTR TrustedDomainName;
    LPWSTR Password;
    BOOL bTrustVerified;
    NET_API_STATUS nas;
    NTSTATUS Status;

    if(argc < 3) {
        fprintf(stderr,
            "Usage: %ls <NewTrustedDomain> <Password> [TargetDomain]\n",
            argv[0]);
        return RTN_USAGE;
    }

    TrustedDomainName = argv[1];    // existing or new TrustedDomain
    Password = argv[2];             // new password for TrustedDomain

    //
    // if a TargetDomain was specified, point to that.
    // if not, set TargetDomain to the current domain if a DC.
    //
    if(argc == 4 && *argv[3] != L'\0') {
        TargetDomainName = argv[3]; // domain to update trust at
        bAllocTargetDomain = FALSE; // no memory allocated
    }
    else {
        PSERVER_INFO_101 si101;
        DWORD Type;
        PUSER_MODALS_INFO_2 umi2;

        //
        // ensure that the local computer is a DC. This operation is only
        // appropriate against a domain controller.
```

```
    //
    nas = NetServerGetInfo(NULL, 101, (LPBYTE *)&si101);
    if(nas != NERR_Success) {
        DisplayError("NetServerGetInfo", nas);
        return RTN_ERROR;
    }

    Type = si101->sv101_type;
    NetApiBufferFree(si101);

    if( !(Type & SV_TYPE_DOMAIN_CTRL) &&
        !(Type & SV_TYPE_DOMAIN_BAKCTRL) ) {
        printf("Error:  Specify a TargetDomain; this operation"
                " is only valid against a domain.\n");
        return RTN_ERROR;
    }

    //
    // obtain the local computer's domain name
    //
    nas = NetUserModalsGet(NULL, 2, (LPBYTE *)&umi2);

    if(nas != NERR_Success) {
        DisplayError("NetUserModalsGet", nas);
        return RTN_ERROR;
    }

    //
    // copy the domain name to new storage
    //
    TargetDomainName = (LPWSTR)HeapAlloc(GetProcessHeap(), 0,
        (lstrlenW(umi2->usrmod2_domain_name) + 1) * sizeof(WCHAR));

    if(TargetDomainName != NULL) {
        lstrcpyW(TargetDomainName, umi2->usrmod2_domain_name);
        bAllocTargetDomain = TRUE; // we allocated memory
    }

    NetApiBufferFree(umi2); // free memory allocated by NetXxx

    //
    // if an error occurred allocating memory, exit
    //
    if(TargetDomainName == NULL) {
        DisplayError("HeapAlloc", 0);
        return RTN_ERROR;
    }
}

//
// do not allow a Domain to trust itself
//
if(lstrcmpiW(TargetDomainName, TrustedDomainName) == 0) {
    fprintf(stderr,"Error:  Domain %ls cannot trust itself.\n",
        TargetDomainName);
```

```
    return RTN_ERROR;
}

//
// ensure Password and TrustedDomainName are the correct length
//
if(lstrlenW(Password) > LM20_PWLEN)
    Password[LM20_PWLEN] = L'\0'; // truncate

if(lstrlenW(TrustedDomainName) > MAX_COMPUTERNAME_LENGTH)
    TrustedDomainName[MAX_COMPUTERNAME_LENGTH] = L'\0'; // truncate

//
// obtain the primary DC computer name from the specified
// TargetDomainName
//
nas = NetGetDCName(
    NULL, TargetDomainName, (LPBYTE *)&DomainController);

if(nas != NERR_Success) {
    DisplayError("NetGetDCName", nas);
    return RTN_ERROR;
}

//
// verify the trust relationship
//
if(!VerifyTrustRelationship(
    TrustedDomainName,
    TargetDomainName,   // trust account to verify
    Password,
    &bTrustVerified
    )) {
    //
    // an error occurred during trust relationship verification
    //
    DisplayError("VerifyTrustRelationship", GetLastError());
}
else {
    //
    // inform the user if the trust was not verified
    //
    if(!bTrustVerified) {
        DWORD dwTrustVerifyFailReason = GetLastError();

        //
        // You could exit here, but this is optional. Reasons for
        // trust verification failure could be non-existent
        // "permitted to trust" on the verified domain (rc=1317),
        // wrong password (rc=1326), etc
        //
        printf("Warning:  ");

        switch(dwTrustVerifyFailReason) {
```

```
            case ERROR_NO_SUCH_USER:
                printf("%ls not permitted-to-trust on %ls\n",
                    TargetDomainName, TrustedDomainName);
                break;

            case ERROR_LOGON_FAILURE:
                printf("Trust %ls has incorrect password on %ls\n",
                    TrustedDomainName, TargetDomainName);
                break;

            default:
                DisplayError("Trust was not verified.  Non-fatal",
                    dwTrustVerifyFailReason);
        } // switch
    }
}

//
// fetch the DomainSid of the domain to trust
//
if(!GetDomainSid(TrustedDomainName, &DomainSid)) {
    DisplayError("GetDomainSid", GetLastError());
    return RTN_ERROR;
}

//
// open the policy on the target domain
//
Status = OpenPolicy(
            DomainController,
            POLICY_CREATE_SECRET |  // for password set operation
            POLICY_TRUST_ADMIN,     // for trust creation
            &PolicyHandle
            );

if(Status != STATUS_SUCCESS) {
    DisplayNtStatus("OpenPolicy", Status);
    return RTN_ERROR;
}

//
// Update TrustedDomainInfo to reflect the specified trust.
//
Status = SetTrustedDomainInfo(
    PolicyHandle,
    DomainSid,
    TrustedDomainName,
    Password
    );

if(Status != STATUS_SUCCESS) {
    DisplayNtStatus("SetTrustedDomainInfo", Status);
    return RTN_ERROR;
}
```

```
    //
    // if you allocated memory for TargetDomainName, free it
    //
    if(bAllocTargetDomain)
        HeapFree(GetProcessHeap(), 0, TargetDomainName);

    //
    // free the Sid which was allocated for the TrustedDomain Sid
    //
    FreeSid(DomainSid);

    //
    // close the policy handle
    //
    LsaClose(PolicyHandle);

    if(DomainController != NULL)
        NetApiBufferFree(DomainController);

    return RTN_OK;
}

void
InitLsaString(
    PLSA_UNICODE_STRING LsaString,
    LPWSTR String
    )
{
    DWORD StringLength;

    if(String == NULL) {
        LsaString->Buffer = NULL;
        LsaString->Length = 0;
        LsaString->MaximumLength = 0;

        return;
    }

    StringLength = lstrlenW(String);
    LsaString->Buffer = String;
    LsaString->Length = (USHORT) StringLength * sizeof(WCHAR);
    LsaString->MaximumLength = (USHORT) (StringLength + 1) *
        sizeof(WCHAR);
}

NTSTATUS
OpenPolicy(
    LPWSTR ServerName,
    DWORD DesiredAccess,
    PLSA_HANDLE PolicyHandle
    )
{
    LSA_OBJECT_ATTRIBUTES ObjectAttributes;
    LSA_UNICODE_STRING ServerString;
    PLSA_UNICODE_STRING Server = NULL;
```

```
    //
    // Always initialize the object attributes to all zeroes
    //
    ZeroMemory(&ObjectAttributes, sizeof(ObjectAttributes));

    if(ServerName != NULL) {
        //
        // Make a LSA_UNICODE_STRING out of the LPWSTR passed in
        //
        InitLsaString(&ServerString, ServerName);

        Server = &ServerString;
    }

    //
    // Attempt to open the policy
    //
    return LsaOpenPolicy(
                Server,
                &ObjectAttributes,
                DesiredAccess,
                PolicyHandle
                );
}

/*++
 This function retrieves the Sid representing the specified DomainName.

 If the function succeeds, the return value is TRUE and pDomainSid will
 point to the Sid representing the specified DomainName. The caller
 should free the memory associated with this Sid with the Win32 API
 FreeSid() when the Sid is no longer needed.

 If the function fails, the return value is FALSE, and pDomainSid will
 be set to NULL.

--*/

BOOL
GetDomainSid(
    LPWSTR DomainName,  // domain name to acquire Sid of
    PSID *pDomainSid    // points to allocated Sid on success
    )
{
    NET_API_STATUS nas;
    LPWSTR DomainController;
    PUSER_MODALS_INFO_2 umi2 = NULL;
    DWORD dwSidSize;
    BOOL bSuccess = FALSE; // assume this function will fail

    *pDomainSid = NULL; // invalidate pointer

    //
    // obtain the PDC computer name of the supplied domain name
```

```
//
nas = NetGetDCName(NULL, DomainName, (LPBYTE *)&DomainController);
if(nas != NERR_Success) {
    SetLastError(nas);
    return FALSE;
}

__try {

//
// obtain the domain Sid from the PDC
// NOTE: this may fail if the credentials of the caller are not
// recognized or valid on the target computer. In this case, it
// is necessary to first establish a connection via NetUseAdd()
// at info-level 2 to the remote IPC$, specifying EMPTY
// credentials.
//
nas = NetUserModalsGet(DomainController, 2, (LPBYTE *)&umi2);
if(nas != NERR_Success) __leave;

//
// if the Sid is valid, obtain the size of the Sid
//
if(!IsValidSid(umi2->usrmod2_domain_id)) __leave;
dwSidSize = GetLengthSid(umi2->usrmod2_domain_id);

//
// allocate storage and copy the Sid
//
*pDomainSid = (PSID)HeapAlloc(GetProcessHeap(), 0, dwSidSize);
if(*pDomainSid == NULL) __leave;

if(!CopySid(dwSidSize, *pDomainSid, umi2->usrmod2_domain_id))
    __leave;

bSuccess = TRUE; // indicate success

} // try
__finally {

//
// free allocated memory
//
NetApiBufferFree(DomainController);

if(umi2 != NULL)
    NetApiBufferFree(umi2);

if(!bSuccess) {
    //
    // if the function failed, free memory and indicate result code
    //

    if(*pDomainSid != NULL) {
        FreeSid(*pDomainSid);
```

```
            *pDomainSid = NULL;
        }

        if(nas != NERR_Success) {
            SetLastError(nas);
        }
    }

    }
    return bSuccess;
}

/*++
 This function manipulates the trust associated with the supplied
 DomainSid.

 If the domain trust does not exist, it is created with the
 specified password. In this case, the supplied PolicyHandle must
 have been opened with POLICY_TRUST_ADMIN and POLICY_CREATE_SECRET
 access to the policy object.

 If the domain trust exists, the password is updated with the
 password specified by the Password parameter. The trust OldPassword
 is set to the previous password.

 If DomainName and Password are NULL, the trusted domain specified by
 DomainSid is deleted.

--*/

NTSTATUS
SetTrustedDomainInfo(
    LSA_HANDLE PolicyHandle,
    PSID DomainSid,             // Sid of domain to manipulate
    LPWSTR TrustedDomainName,   // trusted domain name to add/update
    LPWSTR Password             // new trust password for trusted domain
    )
{
    PTRUSTED_PASSWORD_INFO tpi;
    PTRUSTED_DOMAIN_NAME_INFO tdni;
    LSA_UNICODE_STRING LsaPassword;
    TRUSTED_PASSWORD_INFO Newtpi;
    BOOL bTrustExists = TRUE; // assume trust exists
    NTSTATUS Status;

    //
    // Make sure you were passed a valid Sid
    //
    if(DomainSid == NULL || !IsValidSid(DomainSid))
        return STATUS_INVALID_SID;

    //
    // if TristedDomainName and Password is NULL (or emptry strings),
    // delete the specified trust
    //
```

```
if( (TrustedDomainName == NULL || *TrustedDomainName == L'\0') &&
    (Password == NULL || *Password == L'\0') ) {
    return LsaDeleteTrustedDomain(PolicyHandle, DomainSid);
}

//
// query the current password, which is used to update the
// OldPassword. If the trust doesn't exist, indicate this so that the
// trust will be created.
//
Status = LsaQueryTrustedDomainInfo(
    PolicyHandle,
    DomainSid,
    TrustedPasswordInformation,
    &tpi);

if(Status != STATUS_SUCCESS) {
    if(Status == STATUS_OBJECT_NAME_NOT_FOUND) {
        bTrustExists = FALSE; // indicate trust does not exist
    }
    else {
        DisplayNtStatus("LsaQueryTrustedDomainInfo", Status);
        return Status;
    }
}

InitLsaString(&LsaPassword, Password);

//
// set the new password to the supplied password
//
Newtpi.Password = LsaPassword;

if(bTrustExists) {
    //
    // if the trust already existed, set OldPassword to the
    // current password...
    //
    Newtpi.OldPassword = tpi->Password;
}
else {
    LSA_UNICODE_STRING LsaDomainName;
    DWORD cchDomainName; // number of chars in TrustedDomainName

    //
    // if the trust did not exist, set OldPassword to the
    // supplied password...
    //
    Newtpi.OldPassword = LsaPassword;

    InitLsaString(&LsaDomainName, TrustedDomainName);

    //
    // ...convert TrustedDomainName to uppercase...
    //
```

```
        cchDomainName = LsaDomainName.Length / sizeof(WCHAR);
        while(cchDomainName--) {
            LsaDomainName.Buffer[cchDomainName] =
             towupper(LsaDomainName.Buffer[cchDomainName]);
        }

        //
        // ...create the trusted domain object
        //
        Status = LsaSetTrustedDomainInformation(
            PolicyHandle,
            DomainSid,
            TrustedDomainNameInformation,
            &LsaDomainName
            );

        if(Status != STATUS_SUCCESS) return Status;
    }

    //
    // update TrustedPasswordInformation for the specified trust
    //
    Status = LsaSetTrustedDomainInformation(
        PolicyHandle,
        DomainSid,
        TrustedPasswordInformation,
        &Newtpi
        );

    //
    // if a trust already existed, free the memory associated with
    // the retrieved information
    //
    if(bTrustExists) LsaFreeMemory(tpi);

    return Status;
}

/*++

 This function verifies that the "permitted to trust" side of the
 trust relationship has been established and that the specified
 password matches the "permitted to trust" password.

 If the function fails, the return value is FALSE, and bTrustVerified
 is undefined.

 If the function succeeds, the trust verification succeeded, and
 bTrustVerified is set to the result of the verification.

 If bTrustVerified is TRUE, the trust relationship is verfied.

 If bTrustVerified is FALSE, the trust was not verified.  Call
 GetLastError() to get extended error information.
```

```
 ERROR_NO_SUCH_USER indicates that the "permitted to trust"
 UF_INTERDOMAIN_TRUST_ACCOUNT does not exist on the specified domain.

 ERROR_LOGON_FAILURE indicates that the supplied password does
 not match the password in the "permitted to trust"
 UF_INTERDOMAIN_TRUST_ACCOUNT.

 NOTE: if a connection exists to the domain controller of the
 TargetDomainName prior to calling this function, this function will
 fail due to a credential conflict (WinError == 1219).

--*/

BOOL
VerifyTrustRelationship(
    LPWSTR TargetDomainName,        // domain name to verify trust at
    LPWSTR TrustAccountToVerify,    // trusted domain name to verify
    LPWSTR Password,                // password associated with trust
    LPBOOL bTrustVerified           // indicates if trust was verified
    )
{
    NET_API_STATUS nas;
    USE_INFO_2 ui2;
    LPWSTR DomainController;
    LPWSTR szIpc = L"\\IPC$";
    LPWSTR RemoteResource = NULL;
    LPWSTR UserName = NULL;
    BOOL bSuccess = FALSE; // assume this function will fail

    //
    // fetch the computer name of the domain you try to connect to
    //
    nas = NetGetDCName(
        NULL, TargetDomainName, (LPBYTE *)&DomainController);
    if(nas != NERR_Success) {
        SetLastError(nas);
        return FALSE;
    }

    __try {

    //
    // build the \\<DCName>\ipc$ as the remote resource
    //
    RemoteResource = (LPWSTR)HeapAlloc(GetProcessHeap(), 0,
        (lstrlenW(DomainController) + lstrlenW(szIpc) + 1 ) *
            sizeof(WCHAR)
        );

    if(RemoteResource == NULL) __leave;

    if(lstrcpyW(RemoteResource, DomainController) == NULL) __leave;
    if(lstrcatW(RemoteResource, szIpc) == NULL) __leave;

    //
```

```
// build the user name as <TrustAccountToVerify>$
//
UserName = (LPWSTR)HeapAlloc(GetProcessHeap(), 0,
    (lstrlenW(TrustAccountToVerify) + 1 + 1) * sizeof(WCHAR)
    );

if(UserName == NULL) __leave;

if(lstrcpyW(UserName, TrustAccountToVerify) == NULL) __leave;
if(lstrcatW(UserName, L"$") == NULL) __leave;

ZeroMemory(&ui2, sizeof(ui2));

ui2.ui2_local = NULL;
ui2.ui2_remote = RemoteResource;
ui2.ui2_asg_type = USE_IPC;
ui2.ui2_password = Password;
ui2.ui2_username = UserName;
ui2.ui2_domainname = TargetDomainName;

//
// Attempt to establish a connection to the target domain.
// If the result is ERROR_NOLOGON_INTERDOMAIN_TRUST_ACCOUNT,
// this illustrates the supplied Password matches an existing
// trust account because the credentials were validated but
// a logon is not allowed for this account type.
//
nas = NetUseAdd(NULL, 2, (LPBYTE)&ui2, NULL);

if(nas == ERROR_NOLOGON_INTERDOMAIN_TRUST_ACCOUNT) {
    *bTrustVerified = TRUE;     // indicate trust verified
}
else {
    *bTrustVerified = FALSE;    // indicate trust not verified

    //
    // if the connection succeeded, the UF_INTERDOMAIN_TRUST_ACCOUNT
    // does not exist, because the supplied credentials aren't
    // recognized by the remote (default credentials are applied).
    // Delete the connection and indicate the "permitted to trust"
    // account is non-existent.
    //
    if(nas == NERR_Success) {
        NetUseDel(NULL, RemoteResource, 0);
        nas = ERROR_NO_SUCH_USER;
    }
}

bSuccess = TRUE; // indicate this function succeeded

} // try
__finally {

//
// free allocated memory
```

```
    //
    NetApiBufferFree(DomainController);

    if(RemoteResource != NULL) {
        HeapFree(GetProcessHeap(), 0, RemoteResource);
    }

    if(UserName != NULL) {
        HeapFree(GetProcessHeap(), 0, UserName);
    }

    //
    // if you succeeded, but trust could not be verified, indicate why
    //
    if(bSuccess == TRUE && *bTrustVerified == FALSE) {
        SetLastError(nas);
    }

    }

    return bSuccess;
}

void
DisplayNtStatus(
    LPSTR szAPI,    // ANSI string containing API name
    NTSTATUS Status
    )
{
    //
    // convert the NTSTATUS to Winerror and display the result
    //
    DisplayError(szAPI, LsaNtStatusToWinError(Status));
}

void
DisplayError(
    LPSTR szAPI, // pointer to failed API name
    DWORD dwLastError
    )
{
    LPSTR MessageBuffer;
    DWORD dwBufferLength;

    fprintf(stderr,"%s error! (rc=%lu)\n", szAPI, dwLastError);

    if(dwBufferLength=FormatMessageA(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        dwLastError,
        GetUserDefaultLangID(),
        (LPSTR) &MessageBuffer,
        0,
        NULL
```

```
        ))
    {
        DWORD dwBytesWritten;

        //
        // Output message string on stderr
        //
        WriteFile(
            GetStdHandle(STD_ERROR_HANDLE),
            MessageBuffer,
            dwBufferLength,
            &dwBytesWritten,
            NULL
            );

        //
        // free the buffer allocated by the system
        //
        LocalFree(MessageBuffer);
    }
}
```

Additional reference words: lsa machine account server workstation trust
KBCategory: kbprg kbhowto kbcode
KBSubcategory: BseSecurity NtWkLmapi CodeSam

How to Manage User Privileges Programmatically in Windows NT

PSS ID Number: Q132958

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.51

SUMMARY

In Windows NT, privileges are used to provide a means of access control that differs from discretionary access control. A system manager uses privileges to control which users/groups are able to manipulate various aspects of the system. An application may use privileges when it changes a system resource, such as the system time, or when it shuts down the system.

The User Manager tool can be used to grant and revoke privileges from accounts.

Windows NT 3.51 provides functionality which allows the developer to manage privileges programmatically. This functionality is made available through LSA (Local Security Authority).

An example of an application that would benefit from LSA is a service install program. If the service is configured to run under a user account, it is necessary for that account to have the SeServiceLogonRight "logon as a service" privilege. This article discusses how to take advantage of LSA to grant and revoke privileges from users and groups.

MORE INFORMATION

Managing user privileges can be achieved programmatically using the following steps:

1. Open the policy on the target machine with LsaOpenPolicy(). To grant privileges, open the policy with POLICY_CREATE_ACCOUNT and POLICY_LOOKUP_NAMES access. To revoke privileges, open the policy with POLICY_LOOKUP_NAMES access.

2. Obtain a SID (security identifier) representing the user/group of interest. The LookupAccountName() and LsaLookupNames() APIs can obtain a SID from an account name.

3. Call LsaAddAccountRights() to grant privileges to the user(s) represented by the supplied SID.

4. Call LsaRemoveAccountRights() to revoke privileges from the user(s) represented by the supplied SID.

5. Close the policy with LsaClose().

To successfully grant and revoke privileges, the caller needs to be an
administrator on the target system.

The LSA API LsaEnumerateAccountRights() can be used to determine which
privileges have been granted to an account.

The LSA API LsaEnumerateAccountsWithUserRight() can be used to determine
which accounts have been granted a specified privilege.

Documentation and header files for these LSA APIs is provided in the
Windows 32 SDK in the MSTOOLS\SECURITY directory.

NOTE: These LSA APIs are currently implemented as Unicode only.

This sample will grant the privilege SeServiceLogonRight to the account
specified on argv[1].

This sample is dependant on these import libs

   advapi32.lib (for LsaXxx)
   user32.lib (for wsprintf)

This sample will work properly compiled ANSI or Unicode.

Sample Code

```
/*++

You can use domain\account as argv[1]. For instance, mydomain\scott will
grant the privilege to the mydomain domain account scott.

The optional target machine is specified as argv[2], otherwise, the
account database is updated on the local machine.

The LSA APIs used by this sample are Unicode only.

Use LsaRemoveAccountRights() to remove account rights.

Scott Field (sfield)    12-Jul-95
--*/

#ifndef UNICODE
#define UNICODE
#endif // UNICODE

#include <windows.h>
#include <stdio.h>

#include "ntsecapi.h"

NTSTATUS
OpenPolicy(
    LPWSTR ServerName,          // machine to open policy on (Unicode)
```

```
    DWORD DesiredAccess,        // desired access to policy
    PLSA_HANDLE PolicyHandle    // resultant policy handle
    );

BOOL
GetAccountSid(
    LPTSTR SystemName,          // where to lookup account
    LPTSTR AccountName,         // account of interest
    PSID *Sid                   // resultant buffer containing SID
    );

NTSTATUS
SetPrivilegeOnAccount(
    LSA_HANDLE PolicyHandle,    // open policy handle
    PSID AccountSid,            // SID to grant privilege to
    LPWSTR PrivilegeName,       // privilege to grant (Unicode)
    BOOL bEnable                // enable or disable
    );

void
InitLsaString(
    PLSA_UNICODE_STRING LsaString, // destination
    LPWSTR String                  // source (Unicode)
    );

void
DisplayNtStatus(
    LPSTR szAPI,                // pointer to function name (ANSI)
    NTSTATUS Status             // NTSTATUS error value
    );

void
DisplayWinError(
    LPSTR szAPI,                // pointer to function name (ANSI)
    DWORD WinError              // DWORD WinError
    );

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

//
// If you have the ddk, include ntstatus.h.
//
#ifndef STATUS_SUCCESS
#define STATUS_SUCCESS  ((NTSTATUS)0x00000000L)
#endif

int _cdecl
main(int argc, char *argv[])
{
    LSA_HANDLE PolicyHandle;
    WCHAR wComputerName[256]=L"";   // static machine name buffer
    TCHAR AccountName[256];         // static account name buffer
    PSID pSid;
```

```
NTSTATUS Status;
int iRetVal=RTN_ERROR;          // assume error from main

if(argc == 1)
{
    fprintf(stderr,"Usage: %s <Account> [TargetMachine]\n",
        argv[0]);
    return RTN_USAGE;
}

//
// Pick up account name on argv[1].
// Assumes source is ANSI. Resultant string is ANSI or Unicode
//
wsprintf(AccountName, TEXT("%hS"), argv[1]);

//
// Pick up machine name on argv[2], if appropriate
// assumes source is ANSI. Resultant string is Unicode.
//
if(argc == 3) wsprintfW(wComputerName, L"%hS", argv[2]);

//
// Open the policy on the target machine.
//
if((Status=OpenPolicy(
            wComputerName,      // target machine
            POLICY_CREATE_ACCOUNT | POLICY_LOOKUP_NAMES,
            &PolicyHandle       // resultant policy handle
            )) != STATUS_SUCCESS) {
    DisplayNtStatus("OpenPolicy", Status);
    return RTN_ERROR;
}

//
// Obtain the SID of the user/group.
// Note that we could target a specific machine, but we don't.
// Specifying NULL for target machine searches for the SID in the
// following order: well-known, Built-in and local, primary domain,
// trusted domains.
//
if(GetAccountSid(
        NULL,       // default lookup logic
        AccountName,// account to obtain SID
        &pSid       // buffer to allocate to contain resultant SID
        )) {
    //
    // We only grant the privilege if we succeeded in obtaining the
    // SID. We can actually add SIDs which cannot be looked up, but
    // looking up the SID is a good sanity check which is suitable for
    // most cases.

    //
    // Grant the SeServiceLogonRight to users represented by pSid.
    //
```

```
        if((Status=SetPrivilegeOnAccount(
                    PolicyHandle,           // policy handle
                    pSid,                   // SID to grant privilege
                    L"SeServiceLogonRight", // Unicode privilege
                    TRUE                    // enable the privilege
                    )) == STATUS_SUCCESS)
            iRetVal=RTN_OK;
        else
            DisplayNtStatus("AddUserRightToAccount", Status);
    }
    else {
        //
        // Error obtaining SID.
        //
        DisplayWinError("GetAccountSid", GetLastError());
    }

    //
    // Close the policy handle.
    //
    LsaClose(PolicyHandle);

    //
    // Free memory allocated for SID.
    //
    if(pSid != NULL) HeapFree(GetProcessHeap(), 0, pSid);

    return iRetVal;
}

void
InitLsaString(
    PLSA_UNICODE_STRING LsaString,
    LPWSTR String
    )
{
    DWORD StringLength;

    if (String == NULL) {
        LsaString->Buffer = NULL;
        LsaString->Length = 0;
        LsaString->MaximumLength = 0;
        return;
    }

    StringLength = wcslen(String);
    LsaString->Buffer = String;
    LsaString->Length = (USHORT) StringLength * sizeof(WCHAR);
    LsaString->MaximumLength=(USHORT)(StringLength+1) * sizeof(WCHAR);
}

NTSTATUS
OpenPolicy(
    LPWSTR ServerName,
    DWORD DesiredAccess,
```

```
    PLSA_HANDLE PolicyHandle
    )
{
    LSA_OBJECT_ATTRIBUTES ObjectAttributes;
    LSA_UNICODE_STRING ServerString;
    PLSA_UNICODE_STRING Server = NULL;

    //
    // Always initialize the object attributes to all zeroes.
    //
    ZeroMemory(&ObjectAttributes, sizeof(ObjectAttributes));

    if (ServerName != NULL) {
        //
        // Make a LSA_UNICODE_STRING out of the LPWSTR passed in
        //
        InitLsaString(&ServerString, ServerName);
        Server = &ServerString;
    }

    //
    // Attempt to open the policy.
    //
    return LsaOpenPolicy(
                Server,
                &ObjectAttributes,
                DesiredAccess,
                PolicyHandle
                );
}

/*++
This function attempts to obtain a SID representing the supplied
account on the supplied system.

If the function succeeds, the return value is TRUE. A buffer is
allocated which contains the SID representing the supplied account.
This buffer should be freed when it is no longer needed by calling
HeapFree(GetProcessHeap(), 0, buffer)

If the function fails, the return value is FALSE. Call GetLastError()
to obtain extended error information.

Scott Field (sfield)    12-Jul-95
--*/

BOOL
GetAccountSid(
    LPTSTR SystemName,
    LPTSTR AccountName,
    PSID *Sid
    )
{
    LPTSTR ReferencedDomain=NULL;
    DWORD cbSid=128;    // initial allocation attempt
```

```
DWORD cbReferencedDomain=16; // initial allocation size
SID_NAME_USE peUse;
BOOL bSuccess=FALSE; // assume this function will fail

__try {

//
// initial memory allocations
//
if((*Sid=HeapAlloc(
                GetProcessHeap(),
                0,
                cbSid
                )) == NULL) __leave;

if((ReferencedDomain=HeapAlloc(
                GetProcessHeap(),
                0,
                cbReferencedDomain
                )) == NULL) __leave;

//
// Obtain the SID of the specified account on the specified system.
//
while(!LookupAccountName(
                SystemName,         // machine to lookup account on
                AccountName,        // account to lookup
                *Sid,               // SID of interest
                &cbSid,             // size of SID
                ReferencedDomain,   // domain account was found on
                &cbReferencedDomain,
                &peUse
                )) {
    if (GetLastError() == ERROR_INSUFFICIENT_BUFFER) {
        //
        // reallocate memory
        //
        if((*Sid=HeapReAlloc(
                    GetProcessHeap(),
                    0,
                    *Sid,
                    cbSid
                    )) == NULL) __leave;

        if((ReferencedDomain=HeapReAlloc(
                    GetProcessHeap(),
                    0,
                    ReferencedDomain,
                    cbReferencedDomain
                    )) == NULL) __leave;
    }
    else __leave;
}

//
```

```
    // Indicate success.
    //
    bSuccess=TRUE;

    } // finally
    __finally {

    //
    // Cleanup and indicate failure, if appropriate.
    //

    HeapFree(GetProcessHeap(), 0, ReferencedDomain);

    if(!bSuccess) {
        if(*Sid != NULL) {
            HeapFree(GetProcessHeap(), 0, *Sid);
            *Sid = NULL;
        }
    }

    } // finally

    return bSuccess;
}

NTSTATUS
SetPrivilegeOnAccount(
    LSA_HANDLE PolicyHandle,    // open policy handle
    PSID AccountSid,            // SID to grant privilege to
    LPWSTR PrivilegeName,       // privilege to grant (Unicode)
    BOOL bEnable                // enable or disable
    )
{
    LSA_UNICODE_STRING PrivilegeString;

    //
    // Create a LSA_UNICODE_STRING for the privilege name.
    //
    InitLsaString(&PrivilegeString, PrivilegeName);

    //
    // grant or revoke the privilege, accordingly
    //
    if(bEnable) {
        return LsaAddAccountRights(
                PolicyHandle,       // open policy handle
                AccountSid,         // target SID
                &PrivilegeString,   // privileges
                1                   // privilege count
                );
    }
    else {
        return LsaRemoveAccountRights(
                PolicyHandle,       // open policy handle
                AccountSid,         // target SID
```

```
                FALSE,              // do not disable all rights
                &PrivilegeString,   // privileges
                1                   // privilege count
                );
    }
}

void
DisplayNtStatus(
    LPSTR szAPI,
    NTSTATUS Status
    )
{
    //
    // Convert the NTSTATUS to Winerror. Then call DisplayWinError().
    //
    DisplayWinError(szAPI, LsaNtStatusToWinError(Status));
}

void
DisplayWinError(
    LPSTR szAPI,
    DWORD WinError
    )
{
    LPSTR MessageBuffer;
    DWORD dwBufferLength;

    //
    // TODO: Get this fprintf out of here!
    //
    fprintf(stderr,"%s error!\n", szAPI);

    if(dwBufferLength=FormatMessageA(
                        FORMAT_MESSAGE_ALLOCATE_BUFFER |
                        FORMAT_MESSAGE_FROM_SYSTEM,
                        NULL,
                        WinError,
                        GetUserDefaultLangID(),
                        (LPSTR) &MessageBuffer,
                        0,
                        NULL
                        ))
    {
        DWORD dwBytesWritten; // unused

        //
        // Output message string on stderr.
        //
        WriteFile(
            GetStdHandle(STD_ERROR_HANDLE),
            MessageBuffer,
            dwBufferLength,
            &dwBytesWritten,
            NULL
```

```
            );

        //
        // Free the buffer allocated by the system.
        //
        LocalFree(MessageBuffer);
    }
}
```

Additional reference words: Privilege User Right
KBCategory: kbprg
KBSubcategory: BseSecurity CodeSam

How to Minimize Memory Allocations for New TreeView Control

PSS ID Number: Q130697

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.51 and 4.0

SUMMARY

=======

You can use the new TreeView Common Control to display a hierarchical list of items. This new control is available for Win32-based applications running under Windows NT or Windows 95. Applications typically use this control to display a list of items like directories or files on a given drive. Each node in a TreeView control allocates about 40 bytes. If the TreeView control displays a lot of items, applications can easily consume large amounts of memory, which slows down other applications.

Below are some techniques applications can use to minimize memory allocations for TreeView controls.

MORE INFORMATION

================

TreeView controls maintain internal data structures for every node added to the control. This data structure along with image lists associated with items and text strings for items can drain the physical memory available on the system.

Applications that need to display thousands of items or nodes in the TreeView control can be more proficient about memory allocations by doing the following:

1. When inserting an item into a TreeView control, ensure that the pszText member of the TV_ITEM is not the actual string that needs to be displayed, but is the value LPSTR_TEXTCALLBACK. If a string pointer is passed, the control stores that string internally by allocating memory for it. When this flag is specified, the parent window of the control is responsible for storing the name (string). The string in most cases can be generated dynamically. In this case, the TreeView control sends the parent window a TVN_GETDISPINFO notification message when it needs the item text for displaying, sorting, or editing and sends a TVN_SETDISPINFO notification message when the item text changes.

2. Fill the TreeView nodes on demand. One way to really minimize memory usage in a TreeView control is to fill in only the visible nodes. The TV_ITEM struct's cChildren member can be put to good use for this purpose. This is used as a flag to indicate whether the item has associated child items. It is 1 if the item has one or more child items; otherwise, it is 0 (zero). When inserting visible items into the TreeView control, set this cChildren member to 1 if that node will have child items under it. Do not insert the child items. When the user

clicks the node, the application receives a TVN_ITEMEXPANDING with NM_TREEVIEW.action set to TVE_EXPAND. Insert the child items at that point. Then when the user clicks the same node again (to collapse the node), the applciation receives a TVN_ITEMEXPANDED with NM_TREEVIEW.action set to TVE_COLLAPSE. At that time, collapse the node and all its child items by calling TreeView_Expand (hWndTrevview, hItem, TVE_COLLAPSE|TVE_COLLAPSERESET). This frees up the memory used up by all the children or child items.

3. If the application uses different icons for each child item in the TreeView control, specify the I_IMAGECALLBACK value for the iImage and iSeletedImage members of the TV_ITEM Structure. This way, the control doesn't have to store these images for every child item - thereby reducing the memory requirements for the control as a whole.

Additional reference words: 4.00 usage user styles
KBCategory: kbprg
KBSubcategory: TreeView

How to Modify Executable Code in Memory

PSS ID Number: Q127904

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Follow the steps in this article to create self-modifying code; that is, to modify code pages while they are in memory and execute them there.

NOTE: Self-modifying code is not advised, but there are cases where you may wish to use it.

Step-by-Step Example

1. Call VirtualProtect() on the code pages you want to modify, with the PAGE_WRITECOPY protection.

2. Modify the code pages.

3. Call VirtualProtect() on the modified code pages, with the PAGE_EXECUTE protection.

4. Call FlushInstructionCache().

All four steps are required. The reason for calling FlushInstructionCache() is to make sure that your changes are executed. As processors get faster, the instruction caches on the chips get larger. This allows more out of order prefetching to be done. If you modify your code, but do not call FlushInstructionCache(), the previous instructions may already be in the cache and your changes will not be executed.

Additional reference words: 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMm

How to Modify Printer Settings by Using SetPrinter

PSS ID Number: Q140285

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, 4.0

SUMMARY
=======

SetPrinter is a new API for Windows 95 and Windows NT that allows applications to change various printer attributes. However, as the code in this article demonstrates, a certain amount of preparation is necessary in order to call SetPrinter correctly.

MORE INFORMATION
================

The parameters to SetPrinter() are as follows:

hPrinter

The first parameter is a handle to the printer whose settings are to be changed. This should be retrieved from OpenPrinter().

dwLevel

The second parameter specifies the structure of the data being passed to SetPrinter(). For Windows 95, this can be 0, 2, 3, 4, or 5. For Windows NT, this can be 0, 2, or 3. These numbers correspond to the data type (PRINTER_INFO_n) passed via the third parameter.

lpbPrinter

The third parameter is a PRINTER_INFO_n structure where n corresponds to the number in the second parameter. This structure can cause confusion because it isn't simply a buffer of the size of the structure. These structures contain device-independent information but are immediately followed in memory by some variable amount of device-dependent information, which is given by the device driver. Therefore, a little work is involved to determine how big this buffer should be. This is achieved by calling GetPrinter(), which will set pcbNeeded to the total size needed.

Also, the buffer typically has a large amount of device-independent and device-dependent information in it. Your application is not going to know or care about the values in most of these structure members. So, when you make the changes in which you are interested, you must plug in the correct values for all of these other pieces of data. These other pieces of data are set when you call GetPrinter() a second time.

dwCommand

The fourth parameter is used to pause printing, resume printing, or clear all print jobs. This is typically not used at the same time as lpbPrinter is used. This article is not concerned with setting the printer state, so the sample code sets this parameter to zero.

About DEVMODE

Often, an element of the DEVMODE structure pointed to by pDevMode will be modified (instead of an element of PRINTER_INFO_n). When this is the case, the pDevMode->dmFields flags will tell the application which fields can be changed. Because this is given to you by GetPrinter(), you can check the dmFields flag before attempting the change.

Also, because modifying fields in the device-independent part of DEVMODE may also effect changes in the device-dependent part, you need to call DocumentProperties() before calling SetPrinter() in order to make a consistent DEVMODE structure for SetPrinter().

Sample Code

```
HGLOBAL hGlobal = NULL;
HANDLE hPrinter = NULL;
DWORD dwNeeded = 0;
PRINTER_INFO_2 *pi2 = NULL;
PRINTER_DEFAULTS pd;
BOOL bFlag;
LONG lFlag;

/* Open printer handle (in Windows NT, you need full-access because you
   will eventually use SetPrinter) */

ZeroMemory(&pd, sizeof(pd));
pd.DesiredAccess = PRINTER_ALL_ACCESS;
bFlag = OpenPrinter("My Printer", &hPrinter, &pd);
if (!bFlag || (hPrinter== NULL))
   goto ABORT;

/* The first GetPrinter() tells you how big the buffer should be in order
   to hold all of PRINTER_INFO_2. Note that this will usually return FALSE,
   which only means that the buffer (the third parameter) was not filled
   in. You don't want it filled in here. */

GetPrinter(hPrinter, 2, 0, 0, &dwNeeded);
if (dwNeeded == 0)
   goto ABORT;

/* Allocate enough space for PRINTER_INFO_2. */

hGlobal = GlobalAlloc(GHND, dwNeeded);
```

```
if (hGlobal == NULL)
   goto ABORT;
pi2 = (PRINTER_INFO_2 *)GlobalLock(hGlobal);
if (pi2 == NULL)
   goto ABORT;

/* The second GetPrinter() fills in all the current settings, so all you
   need to do is modify what you'r interested in. */

bFlag = GetPrinter(hPrinter, 2, (LPBYTE)pi2, dwNeeded, &dwNeeded);
if (!bFlag)
   goto ABORT;

/* Set orientation to Landscape mode if the driver supports it. */

if ((pi2->pDevMode != NULL) && (pi2->pDevMode->dmFields & DM_ORIENTATION))
{
   /* Change the devmode. */
   pi2->pDevMode->dmOrientation = DMORIENT_LANDSCAPE;

   /* Make sure the driver-dependent part of devmode is updated as
      necessary. */
   lFlag = DocumentProperties(hwnd, hPrinter,
            "My Printer",
            pi2->pDevMode, pi2->pDevMode,
            DM_IN_BUFFER | DM_OUT_BUFFER);
   if (lFlag != IDOK)
      goto ABORT;

   /* Update printer information. */
   bFlag = SetPrinter(hPrinter, 2, (LPBYTE)pi2, 0);
   if (!bFlag)
      /* The driver supported the change but wasn't allowed due to some
         other reason (probably lack of permission). */
      goto ABORT;
}
else
   /* The driver doesn't support changing this. */
   goto ABORT;

/* Clean up. */
ABORT:   if (pi2 != NULL)
      GlobalUnlock(hGlobal);
   if (hGlobal != NULL)
      GlobalFree(hGlobal);
   if (hPrinter != NULL)
      ClosePrinter(hPrinter);
```

Additional reference words: 3.50 3.51 4.00 print settings
KBCategory: kbgraphic kbcode kbhowto
KBSubcategory: GdiPrn

How to Modify the Width of the Drop Down List in a Combo Box

PSS ID Number: Q131845

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY
=======

The Windows combo box contains a list box (of the ComboLBox class) within it. In the standard combo box, this list box has exactly the same width as the combo box. However, you can make the width of the list box wider or narrower than the width of the combo box. You may have seen combo box lists like this in Microsoft Word and Microsoft Excel. This article shows by example how to subclass a standard combo box class to achieve this functionality.

MORE INFORMATION
================

The combo box in Windows is actually a combination of two or more controls; that's why it's called a "combo" box. For more information about the parts of a combo box and how they relate to each other, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q65881
   TITLE   : The Parts of a Windows Combo Box and How They Relate

To make the combo box list wider or narrower, you need the handle of the list box control within the combo box. This task is difficult because the list box is actually a child of the desktop window (for CBS_DROPDOWN and CBS_DROPDOWNLIST styles). If it were a child of the ComboBox control, dropping down the list box would clip it to the parent, and it wouldn't display.

A combo box receives WM_CTLCOLOR messages for its component controls when they need to be painted. This allows the combo box to specify a color for these controls. The HIWORD of the lParam in this message is the type of the control. In case of the combo box, Windows sends it a WM_CTLCOLOR message with the HIWORD set to CTLCOLOR_LISTBOX when the list box control needs to be painted. The LOWORD of the lParam contains the handle of the list box control.

In 32-bit Windows, the WM_CTLCOLOR message has been replaced with multiple messages, one for each type of control (WM_CTLCOLORBTN). For more information about this, please see the following article in the Microsoft

Knowledge Base:

   ARTICLE-ID: Q81707
   TITLE   : WM_CTLCOLOR Processing for Combo Boxes of All Styles

Once you obtain the handle to the list box control window, you can resize the control by using the MoveWindow API.

The following code sample demonstrates how to do this. This sample assumes that you have placed the combo box control in a dialog box.

Sample Code

```
// Global declarations.

LRESULT CALLBACK NewComboProc (HWND hWnd,   UINT message,   WPARAM
   wParam, LPARAM lParam ); // prototype for the combo box subclass proc

HANDLE hInst;                     // Current app instance
BOOL bFirst;                      // a flag

// Dialog procedure for the dialog containing the combo box.

BOOL __export CALLBACK DialogProc(HWND hDlg, UINT message, WPARAM wParam,
            LPARAM lParam)
{
    FARPROC lpfnNewComboProc;

    switch (message)
    {
    case WM_INITDIALOG:

   bFirst = TRUE;       // set flag here - see below for usage

   // subclass the combo box

   lpfnOldComboProc = (FARPROC ) SetWindowLong (
               GetDlgItem ( hDlg, IDC_COMBO1 ),
               GWL_WNDPROC,
               (LONG)NewComboProc );
   break;

     case WM_DESTROY:
   (FARPROC ) SetWindowLong (    GetDlgItem ( hDlg, IDC_COMBO1 ),
               GWL_WNDPROC,
               (LONG)lpfnOldComboProc );
   break;
     default:
        break;
     }

     return FALSE;

} // end dialog proc
```

```
// Combobox subclass proc.

LRESULT CALLBACK NewComboProc (HWND hWnd,   UINT message,   WPARAM
            wParam, LPARAM lParam );

{
   static HWND hwndList;
   static RECT rectList;

#ifdef WIN16
   if (  WM_CTLCOLOR  == message)  // combo controls are to be painted.
#else
   if ( WM_CTLCOLORLISTBOX == message ) // 32 bits has new message.
#endif
   {
      // is this message for the list box control in the combo?
#ifdef WIN16
      if ( CTLCOLOR_LISTBOX==HIWORD (lParam )   ) // need only for 16 bits
      {
#endif
         // Do only the very first time, get the list
         // box handle and the list box rectangle.
         // Note the use of GetWindowRect, as the parent
         // of the list box is the desktop window

         if ( bFirst )
         {
#ifdef WIN16
            hwndList = LOWORD (lParam );
#else
            hwndList = (HWND) lParam ;     // HWND is 32 bits
#endif
            GetWindowRect ( hwndList, &rectList );
                        bFirst = FALSE;
         }

         // Resize listbox window cx by 50 ( use your size here )

         MoveWindow ( hwndList, rectList.left, rectList.top,
         ( rectList.right - rectList.left + 50 ),
         rectList.bottom - rectList.top, TRUE );
#ifdef WIN16
      }
#endif
   }

   // Call original combo box procedure to handle other combo messages.

   return CallWindowProc ( lpfnOldComboProc, hWnd, message,
wParam, lParam  );
}
```

Additional reference words: 1.20 3.10 3.50 4.00 95
KBCategory: kbui kbcode
KBSubcategory: UsrCtl

How to Move Files That Are Currently in Use

PSS ID Number: Q140570

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  - Microsoft Windows NT versions 3.1, 3.5, 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Sometimes Win32 applications need to delete, rename, or move files that are currently being used by the system. One common example is that setup programs need to remove themselves from the user's hard disk when they are finished setting up a software package. Sometimes, they also need to move device drivers that are currently being used by the system. Applications need help from the operating system to delete or move these files.

Windows 95 and Windows NT each provide a unique method for helping applications to remove, replace, or rename files and directories that are in use. Although the two platforms differ in how they implement these methods, both share an overall strategy where the application specifies which files to process, and the system processes them when it reboots. This article explains how applications can use the method provided by each Windows platform.

MORE INFORMATION

Moving Files in Windows NT

Win32-based applications running in Windows NT should use MoveFileEx() with the MOVEFILE_DELAY_UNTIL_REBOOT flag to move, replace, or delete files and directories currently being used. The next time the system is rebooted, the Windows NT bootup program will move, replace, or delete the specified files and directories.

To move or replace a file or directory that is in use, an application must specify both a source and destination path on the same volume (for example, drive C:). If the destination path is an existing file, it will be overwritten. If the destination path is an existing directory, it will not be overwritten and both the source and destination paths will remain unchanged. Here is an example call to move or replace a file or move a directory:

```
// Move szSrcFile to szDstFile next time system is rebooted
MoveFileEx(szSrcFile, szDstFile, MOVEFILE_DELAY_UNTIL_REBOOT);
```

To delete a file or directory, the application must set the destination path to NULL. If the source path is a directory, it will be removed only if

it is empty. Note that if you must use MoveFileEx() to remove files from a directory, you must reboot the computer before you can call MoveFileEx() to remove the directory. Here is an example of how to delete a file or empty a directory:

```
// Delete szSrcFile next time system is rebooted
MoveFileEx(szSrcFile, NULL, MOVEFILE_DELAY_UNTIL_REBOOT);
```

Moving Files in Windows 95

Windows 95 does not implement MoveFileEx() but does provide an alternate way for all Win32-based, 16-bit Windows-based, and MS-DOS-based applications to move, replace, or delete files (but not directories) that are currently in use. This capability is implemented through the [rename] section of a file named Wininit.ini. If Wininit.ini is present in the Windows directory, Wininit.exe processes it when the system boots. Once Wininit.ini has been processed, Wininit.exe renames it to Wininit.bak.

The syntax of the [rename] section is:

```
DestinationFileName=SourceFileName
```

DestinationFileName and SourceFileName must be short (8.3) file names because Wininit.ini is processed before the protected mode disk system is loaded, and long file names are only available when the protected mode disk system is running. Destination and source files specified in Wininit.ini with long file names are ignored.

The [rename] section can have multiple lines with one file per line. To delete a file, specify NUL as the DestinationFileName. Here are some example entries:

```
[rename]
NUL=C:\TEMP.TXT
C:\NEW_DIR\EXISTING.TXT=C:\EXISTING.TXT
C:\NEW_DIR\NEWNAME.TXT=C:\OLDNAME.TXT
C:\EXISTING.TXT=C:\TEMP\NEWFILE.TXT
```

The first line causes Temp.txt to be deleted. The second causes Existing.txt to be moved to a new directory. The third causes Oldname.txt to be moved and renamed. The fourth causes an existing file to be overwritten by Newfile.txt.

Applications should not use WritePrivateProfileString() to write entries to the [rename] section because there can be multiple lines with the same DestinationFileName, especially if DestinationFileName is "NUL." Instead, they should add entries by parsing Wininit.ini and appending the entries to the end of the [rename] section.

NOTE: Always use a case-insensitive search to parse Wininit.ini because the title of the [rename] section and the file names inside it may have any combination of uppercase and lowercase letters.

Applications that use Wininit.ini should check for its existence in the

Windows directory. If Wininit.ini is present, then another application has written to it since the system was last restarted. Therefore, the application should open it and add entries to the [rename] section. If Wininit.ini isn't present, the application should create it and add to the [rename] section. Doing so ensures that entries from other applications won't be deleted accidentally by your application.

To undo a file rename operation before the system is rebooted, you must remove the corresponding line from the [rename] section of the Wininit.ini file.

Additional reference words: 3.51 4.00 update install setup
KBCategory: kbprg kbhowto
KBSubcategory: BseFileio

How to Obtain a Handle to Any Process with SeDebugPrivilege

PSS ID Number: Q131065

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY
=======

In Windows NT, you can retrieve a handle to any process in the system by enabling the SeDebugPrivilege in the calling process. The calling process can then call the OpenProcess() Win32 API to obtain a handle with PROCESS_ALL_ACCESS.

MORE INFORMATION
================

This functionality is provided for system-level debugging purposes. For debugging non-system processes, it is not necessary to grant or enable this privilege.

This privilege allows the caller all access to the process, including the ability to call TerminateProcess(), CreateRemoteThread(), and other potentially dangerous Win32 APIs on the target process.

Take great care when granting SeDebugPrivilege to users or groups.

Sample Code

The following source code illustrates how to obtain SeDebugPrivilege in order to get a handle to a process with PROCESS_ALL_ACCESS. The sample code then calls TerminateProcess on the resultant process handle.

```
/*++

The SeDebugPrivilege allows you to open any process for debugging purposes.
After enabling the privilege, you can open a target process by using
OpenProcess() with PROCESS_ALL_ACCESS.

By default, this privilege is granted only to SYSTEM and the local
Administrators group.

User Manager | Policies | User Rights | Show Advanced User Rights | Debug
Programs can be used to grant or revoke this privilege to arbitrary users
or groups.

WARNING: This privilege allows all access to a process. A malevolent user
could open a system process, create a remote thread in the system process,
```

```
and execute code in the system security context. Great care must be used
when giving out this privilege

--*/

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

#include <windows.h>
#include <stdio.h>

BOOL SetPrivilege(
    HANDLE hToken,          // token handle
    LPCTSTR Privilege,      // Privilege to enable/disable
    BOOL bEnablePrivilege   // TRUE to enable.  FALSE to disable
    );

void DisplayError(LPTSTR szAPI);

int main(int argc, char *argv[])
{
    HANDLE hProcess;
    HANDLE hToken;
    int dwRetVal=RTN_OK; // assume success from main()

    // show correct usage for kill
    if (argc != 2)
    {
        fprintf(stderr,"Usage: %s [ProcessId]\n", argv[0]);
        return RTN_USAGE;
    }

    if(!OpenProcessToken(
            GetCurrentProcess(),
            TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
            &hToken
            )) return RTN_ERROR;

    // enable SeDebugPrivilege
    if(!SetPrivilege(hToken, SE_DEBUG_NAME, TRUE))
    {
        DisplayError("SetPrivilege");

        // close token handle
        CloseHandle(hToken);

        // indicate failure
        return RTN_ERROR;
    }

   // open the process
    if((hProcess = OpenProcess(
            PROCESS_ALL_ACCESS,
            FALSE,
```

```
            atoi(argv[1]) // PID from commandline
            )) == NULL)
    {
        DisplayError("OpenProcess");
        return RTN_ERROR;
    }

    // disable SeDebugPrivilege
    SetPrivilege(hToken, SE_DEBUG_NAME, FALSE);

    if(!TerminateProcess(hProcess, 0xffffffff))
    {
        DisplayError("TerminateProcess");
        dwRetVal=RTN_ERROR;
    }

    // close handles
    CloseHandle(hToken);
    CloseHandle(hProcess);

    return dwRetVal;
}

BOOL SetPrivilege(
    HANDLE hToken,          // token handle
    LPCTSTR Privilege,      // Privilege to enable/disable
    BOOL bEnablePrivilege   // TRUE to enable.  FALSE to disable
    )
{
    TOKEN_PRIVILEGES tp;
    LUID luid;
    TOKEN_PRIVILEGES tpPrevious;
    DWORD cbPrevious=sizeof(TOKEN_PRIVILEGES);

    if(!LookupPrivilegeValue( NULL, Privilege, &luid )) return FALSE;

    //
    // first pass.  get current privilege setting
    //
    tp.PrivilegeCount           = 1;
    tp.Privileges[0].Luid       = luid;
    tp.Privileges[0].Attributes = 0;

    AdjustTokenPrivileges(
            hToken,
            FALSE,
            &tp,
            sizeof(TOKEN_PRIVILEGES),
            &tpPrevious,
            &cbPrevious
            );

    if (GetLastError() != ERROR_SUCCESS) return FALSE;

    //
```

```
    // second pass.  set privilege based on previous setting
    //
    tpPrevious.PrivilegeCount       = 1;
    tpPrevious.Privileges[0].Luid   = luid;

    if(bEnablePrivilege) {
        tpPrevious.Privileges[0].Attributes |= (SE_PRIVILEGE_ENABLED);
    }
    else {
        tpPrevious.Privileges[0].Attributes ^= (SE_PRIVILEGE_ENABLED &
            tpPrevious.Privileges[0].Attributes);
    }

    AdjustTokenPrivileges(
            hToken,
            FALSE,
            &tpPrevious,
            cbPrevious,
            NULL,
            NULL
            );

    if (GetLastError() != ERROR_SUCCESS) return FALSE;

    return TRUE;
}

void DisplayError(
    LPTSTR szAPI    // pointer to failed API name
    )
{
    LPTSTR MessageBuffer;
    DWORD dwBufferLength;

    fprintf(stderr,"%s() error!\n", szAPI);

    if(dwBufferLength=FormatMessage(
                FORMAT_MESSAGE_ALLOCATE_BUFFER |
                FORMAT_MESSAGE_FROM_SYSTEM,
                NULL,
                GetLastError(),
                GetSystemDefaultLangID(),
                (LPTSTR) &MessageBuffer,
                0,
                NULL
                ))
    {
        DWORD dwBytesWritten;

        //
        // Output message string on stderr
        //
        WriteFile(
                GetStdHandle(STD_ERROR_HANDLE),
                MessageBuffer,
```

```
                dwBufferLength,
                &dwBytesWritten,
                NULL
                );

        //
        // free the buffer allocated by the system
        //
        LocalFree(MessageBuffer);
    }
}
```

Additional reference words: 3.10 3.50 3.51 OpenProcess TerminateProcess
KBCategory: kbprg kbcode
KBSubcategory: BseSecurity BseMisc CodeSam

How to Obtain Filename and Path from a Shell Link or Shortcut

PSS ID Number: Q130698

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.51 and 4.0

SUMMARY
=======

The shortcuts used in Microsoft Windows 95 provide applications and users a way to create shortcuts or links to objects in the shell's namespace. The IShellLink OLE Interface can be used to obtain the path and filename from the shortcut, among other things.

MORE INFORMATION
================

A shortcut allows the user or an application to access an object from anywhere in the namespace. Shortcuts to objects are stored as binary files. These files contain information such as the path to the object, working directory, the path of the icon used to display the object, the description string, and so on.

Given a shortcut, applications can use the IShellLink interface and its functions to obtain all the pertinent information about that object. The IShellLink interface supports functions such as GetPath(), GetDescription(), Resolve(), GetWorkingDirectory(), and so on.

Sample Code

The following code shows how to obtain the filename or path and description of a given link file:

```
#include <windows.h>
#include <shlobj.h>

// GetLinkInfo() fills the filename and path buffer
// with relevant information.
// hWnd         - calling application's window handle.
//
// lpszLinkName - name of the link file passed into the function.
//
// lpszPath     - the buffer that receives the file's path name.
//
// lpszDescription - the buffer that receives the file's
// description.
HRESULT
GetLinkInfo( HWND    hWnd,
             LPCTSTR lpszLinkName,
             LPSTR   lpszPath,
             LPSTR   lpszDescription)
```

```
{

   HRESULT hres;
   IShellLink *pShLink;
   WIN32_FIND_DATA wfd;

// Initialize the return parameters to null strings.
   *lpszPath = '\0';
   *lpszDescription = '\0';

// Call CoCreateInstance to obtain the IShellLink
// Interface pointer. This call fails if
// CoInitialize is not called, so it is assumed that
// CoInitialize has been called.
   hres = CoCreateInstance( &CLSID_ShellLink,
                            NULL,
                            CLSCTX_INPROC_SERVER,
                            &IID_IShellLink,
                            (LPVOID *)&pShLink );

   if (SUCCEEDED(hres))
   {
      IPersistFile *ppf;

// The IShellLink Interface supports the IPersistFile
// interface. Get an interface pointer to it.
      hres = pShLink->lpVtbl->QueryInterface(pShLink,
                                        &IID_IPersistFile,
                                        (LPVOID *)&ppf );
      if (SUCCEEDED(hres))
      {
         WORD wsz[MAX_PATH];

// Convert the given link name string to a wide character string.
         MultiByteToWideChar( CP_ACP, 0,
                              lpszLinkName,
                              -1, wsz, MAX_PATH );
// Load the file.
         hres = ppf->lpVtbl->Load(ppf, wsz, STGM_READ );
         if (SUCCEEDED(hres))
         {
// Resolve the link by calling the Resolve() interface function.
// This enables us to find the file the link points to even if
// it has been moved or renamed.
            hres = pShLink->lpVtbl->Resolve(pShLink,  hWnd,
                                            SLR_ANY_MATCH | SLR_NO_UI);
            if (SUCCEEDED(hres))
            {
// Get the path of the file the link points to.
               hres = pShLink->lpVtbl->GetPath( pShLink, lpszPath,
                                            MAX_PATH,
                                            &wfd,
                                            SLGP_SHORTPATH );

// Only get the description if we successfully got the path
```

```
   // (We can't return immediately because we need to release ppf &
   //  pShLink.)
                  if(SUCCEEDED(hres))
                  {
   // Get the description of the link.
                    hres = pShLink->lpVtbl->GetDescription(pShLink,
                                                           lpszDescription,
                                                           MAX_PATH );
                  }
               }
            }
            ppf->lpVtbl->Release(ppf);
         }
         pShLink->lpVtbl->Release(pShLink);
      }
      return hres;
   }
```

Additional reference words: 4.00
KBCategory: kbprg kbcode kbole
KBSubcategory: IShellLink

How to Obtain Fonts, ToolTips, and Other Non-Client Metrics

PSS ID Number: Q130764

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The SystemParametersInfo() API under Windows 95 has been expanded to include a new set of FLAGS to set or get system-wide parameters. One such flag is the SPI_GET/SETNONCLIENTMETRICS flag and the NONCLIENTMETRICS structure. This flag and the related structure when used with the SystemsParametersInfo() API can provide applications with a plethora of information on the non-client metrics system wide.

MORE INFORMATION
================

Windows 95 provides users with the ability to change the fonts used (displayed) by menus and message boxes and change the height of caption bars of windows by simply changing the settings in the Appearance property sheet. They need only right click the desktop to bring up the Display properties dialog box; then they can choose the Appearance page. This was not possible under Windows version 3.1.

Windows 95 applications can programatically change these features with the help of the SystemParametersInfo() function and the NONCLIENTMETRICS structure. Windows 95 applications should not randomly change these system settings unless absolutely necessary because system-wide changes occur.

To obtain the fonts used by message boxes, menus, status bars, and captions, applications can call SystemParametersInfo() with the SPI_GETNONCLIENTMETRICS flag, passing the address of the NONCLIENTMETRICS structure as the third parameter. The system then fills this structure with all sorts of information. The lfMessageFont, lfMenuFont, lfStatusFont members of the NONCLIENTMETRICS structure have the font information.

Similarly, applications can change the font used by menus, message boxes, status bars, and captions by calling SystemParametersInfo() with the SPI_SETNONCLIENTMETRICS flag. When this flag is spcified, a NONCLIENTMETRICS structure is filled with the appropriate values if its address is passed in as the third parameter. Once again, the changes made this way are reflected system wide, so application designers should use this flag sparingly.

Additional reference words: 4.00 user controls styles
KBCategory: kbui
KBSubcategory: UsrSys

How to Obtain Icon Information from an .EXE in Windows 95

PSS ID Number: Q131500

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

SHGetFileInfo is a new API that Windows 95 provides to allow an application to extract icon information from a particular file. With the introduction of large (32x32) and small (16x16) icons in Windows 95, SHGetFileInfo provides this information as well by filling in the appropriate members of the SHFILEINFO structure.

MORE INFORMATION

================

In Windows version 3.1, an application can use the ExtractIcon() to retrieve the handle of an icon associated with a specified executable file, dynamic link library, or icon file.

Windows 95 provides a new function, SHGetFileInfo(), which, among other things, provides this icon information when using the SHGFI_ICON flag. When the function returns, the handle to the icon is returned in the hIcon member of the SHFILEINFO structure, and the index of the icon within the system image list is returned in the iIcon member of the same structure.

The code below demonstrates how to retrieve the Gen32 sample's icon and associates that icon with a dialog box's button:

```
   HICON  hGen32Icon;
   SHFILEINFO  shfi;

   if (SHGetFileInfo ((LPCSTR)"C:\\MySamplesDir\\Gen32\\Gen32.Exe",
                       0,
                       &shfi,
                       sizeof (SHFILEINFO),
                       SHGFI_ICON))
   {
      hGen32Icon = shfi.hIcon;

      // Note that this button has been defined in the .RC file
      // to be of BS_ICON style
      SendDlgItemMessage (hDlg,
                          IDC_BUTTON,
                          BM_SETIMAGE,
                          (WPARAM)IMAGE_ICON,
                          (LPARAM)(HICON)hGen32Icon);
   }
```

```
   else
   {
   // SHGetFileInfo failed...
   }
```

Windows 95 also introduces the concept of large and small icons associated with applications where the large icon is displayed when the application is minimized and the small icon is displayed in the upper-left corner of the application. This small icon, when clicked, drops down the application's system menu.

SHGetFileInfo() provides the file's large and small icon information as well, using the SHGFI_LARGEICON and SHGFI_SMALLICON flags respectively. SHGFI_LARGEICON returns the handle of the system image list containing the large icon images, whereas SHGFI_SMALLICON returns that of the system image list containing the small icon images. These flags, when OR'ed with the SHGFI_SYSICONINDEX flag, return the icon index within the appropriate system image list in the iIcon member of the SHFILEINFO struct.

The code sample below demonstrates how to retrieve the small icon associated with the same GEN32 sample.

```
   HICON  hGen32Icon;
   HIMAGELIST hSysImageList;
   SHFILEINFO  shfi;

   hSysImageList = SHGetFileInfo
((LPCSTR)"C:\\MySamplesDir\\Gen32\\Gen32.Exe",
                                  0,
                                  &shfi,
                                  sizeof (SHFILEINFO),
                                  SHGFI_SYSICONINDEX | SHGFI_SMALLICON);
   if (hSysImageList)
   {
   hGen32Icon = ImageList_GetIcon (hSysImageList,
                                       shfi.iIcon,
                                       ILD_NORMAL);
   }
   else
   {
   // SHGetFileInfo failed...
   }
```

Before closing, the application must call DestroyIcon() to free system resources associated with the icon returned by ImageList_GetIcon().

NOTE: The SHGetFileInfo() API will be supported in the next release of Windows NT that supports the new shell interface very similar to Windows 95. This API is not supported in Windows NT version 3.51.

Additional reference words: DLL EXE ICO 32 x 32 16 x 16
KBCategory: kbui kbcode
KBSubcategory: UsrRsc

How to Obtain MIDI Specifications

PSS ID Number: Q140203

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Win32s version 1.2

SUMMARY
=======

The Musical Instrument Digital Interface (MIDI) specifications are published by and copyrighted material of the MIDI Manufacturers Association (MMA). Certain of these specifications are useful to Windows developers who author MIDI files for playback or who write software to record MIDI data or play it back without relying on Windows MCI support. However, the specifications are not available from Microsoft. They must be obtained from the MMA or its distributors.

As of November 1995, the MMA provides all MIDI specifications (including several more specialized ones not listed in this article) in a single document. It may be obtained by ordering item #3535 from the Mix Bookshelf, which can be contacted at 800-233-9604. For current information and other questions, please contact the MMA at the following address or phone numbers:

MIDI Manufacturers Association
PO Box 3173
La Habra, CA 90632-3173

voice: 310-947-8689
fax: 310-947-4569

MORE INFORMATION
================

Following are several of the specifications that are of the greatest general interest to multimedia application developers. The term "low-level MIDI" refers to the midiXXX functions, as opposed to MCI services.

Knowledge of these specifications is not necessary to implement playback of MIDI files using MCI services because the MCI driver shields applications from these details.

MIDI Detailed Specification (for Low-Level MIDI Development)

The MIDI Detailed Specification explains the MIDI hardware and software protocols and is of interest to developers of multimedia applications that implement MIDI support using low-level MIDI APIs to record, edit, or play MIDI data.

Standard MIDI Files 1.0 (for Low-Level MIDI Development)

The Standard MIDI Files specification defines a way to interchange time-stamped MIDI data between different applications on the same or different hardware platforms. This is useful to developers writing applications that read and parse disk files containing MIDI data or that write MIDI data files to disk.

General MIDI System - Level 1 (for MIDI File Authoring)

The General MIDI (GM) specification defines a minimum MIDI configuration of a "General MIDI System" consisting of a certain class of MIDI playback devices. It is of interest to multimedia developers who author MIDI files. Most PC sound cards and MIDI synthesizers manufactured today are compatible with the GM specification. MIDI files that are authored to the GM specification should generally sound like they were intended to sound no matter which GM-compatible device they are played on.

Additional reference words: IMA 3.10 4.00 3.50
KBCategory: kbmm kbprg
KBSubcategory: MMMidi

How to Obtain the Video for Windows Version

PSS ID Number: Q140067

The information in this article applies to:

- Microsoft Video for Windows Development Kit version 1.1
- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0

SUMMARY

This article explains how to obtain the version number for Video for Windows.

MORE INFORMATION

Using the GetVersionEx() Function

Video for Windows has been incorporated into Windows 95 and Windows NT version 3.51. Therefore, 32-bit applications should call the GetVersionEx() function to determine the version of the operating system and consequently the version of the multimedia system that is installed. GetVersionEx() returns extended information about the version of the operating system that is currently running.

Using the VideoForWindowsVersion() Function

16-bit applications (for example, those compiled with Visual C++ version 1.5x) should call the VideoForWindowsVersion() function to determine the version of Video for Windows that is installed. The VideoForWindowsVersion() function is not documented in the Video for Windows Development Kit. However, many of the samples from the kit use this function, and it is defined in the multimedia header files. The function is exported by Msvideo.dll, which is usually located in the Windows\System directory.

This function may be used to retrieve the major and minor versions of the currently installed Video for Windows system under Windows 3.x. For example, the following code fragment verifies that Video for Windows version 1.1 or higher is currently running:

```
WORD   wVer;

// First make sure you are running version 1.1 or later
wVer = HIWORD(VideoForWindowsVersion());
if (wVer < 0x010a)
{
   // oops, too old
   MessageBox(NULL, "Video for Windows version is too old",
```

```
            "Error", MB_OK | MB_ICONSTOP);
   return FALSE;
}
```

NOTE: As stated in the Win32 SDK, this function is obsolete in Windows NT and Windows 95. The GetVersion() or GetVersionEx() functions should be used on these platforms rather than VideoForWindowsVersion().

Here is an excerpt from the Win32 SDK "Multimedia, Introduction, Multimedia Possibilities, Version Checking" documentation:

> You may need to check the installed version of the multimedia system, particularly if your application takes advantage of features that were not available in previous releases. Although the multimedia header files contain two version-checking functions, they are obsolete. These obsolete functions are mmsystemGetVersion and VideoForWindowsVersion. Your application should rely on the standard Windows functions, GetVersion or GetVersionEx, instead.

Using the MCI String

The Multimedia Control Interface (MCI) string "info avivideo version" should not be used to determine the installed version of Video for Windows. This string is only meant to be displayed; it is not meant to be used for version comparison. It will become obsolete in the future.

Additional reference words: win16sdk 1.10 3.10 4.00
KBCategory: kbmm kbprg kbdocerr kbhowto
KBSubcategory: MMVideo

How To Open Volumes Under Windows 95

PSS ID Number: Q125712

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

Windows 95 does not support opening disk drives or disk partitions with CreateFile(), as Windows NT does. Windows 95 also does not support the DeviceIoControl() IOCTL APIs, as Windows NT does. Instead, low-level disk access in Windows 95 can be achieved through DeviceIoControl() calls to the VWIN32 VxD.

MORE INFORMATION

Windows NT supports obtaining a handle to a disk drive or disk partition by using CreateFile() and specifying the name of the drive or partition as the filename (e.g. "\\.\PHYSICALDRIVE0" or "\\.\C:"). This handle can then be used in the DeviceIoControl() Win32 API.

Windows 95 differs in the following ways:

1. Obtaining a disk drive or disk partition handle is not supported. The call to CreateFile() will fail, and GetLastError() will return error code 2, ERROR_FILE_NOT_FOUND.

2. The DeviceIoControl IOCTL functions (such as IOCTL_DISK_FORMAT_TRACKS) are not supported. These IOCTLs require the handle to a disk drive or disk partition and thus can't be used.

3. DeviceIoControl() is called using a handle to a VxD rather than a handle to a disk drive or disk partition. Obtain a handle to VWIN32.VXD by using CreateFile("\\\\.\\VWIN32", ...). Use this handle in calls to DeviceIoControl() to perform volume locking (Int 21h Function 440Dh, Subfunctions 4Ah and 4Bh), and then to perform BIOS calls (Int 13h), Absolute Disk Reads/Writes (Int 25h and 26h), or MS-DOS IOCTL functions (Int 21h Function 440Dh).

REFERENCES

For information on using DeviceIoControl() in Windows 95 and the IOCTL functions supported by the VWIN32 VxD, please see the help file "Windows 95 Guide to Programming". Go to "Using Windows 95 features" and select "Using Device I/O Control."

For information on using CreateFile() to obtain disk drive or disk

partition handles under Windows NT, see the description for CreateFile() in the Microsoft Windows Programmer's Reference, Volume 3.

For information on Windows 95 extensions to the MS-DOS interrupts, please see the help file "Windows 95 Guide to Programming." Go to "Using Microsoft MS-DOS Extensions" and select "Exclusive Volume Locking", then "About Exclusive Volume Locking", then "Exclusive Use Lock".

For a complete list of IOCTLs, see the description of the DeviceIoControl() function in the Microsoft Windows Programmer's Reference, Volume 3.

Additional reference words: 4.00
KBCategory: kbprg
KBSubcategory: BseFileio

How to Overlay Images Using Image List Controls

PSS ID Number: Q125629

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

One of the more interesting controls Windows 95 provides as part of its new common controls is the image list. Image lists provide an easy way to manage a group of bitmaps and draw them on the screen, without having to worry about calling CreateCompatibleDC(), SelectObject(), BitBlt(), StretchBlt(), and so on.

One interesting feature that image lists provide through the ImageList_Draw() API is the ability to overlay images -- that is, to draw an image transparently over another image. Calling ImageList_Draw() with the last parameter set to an index to an overlay mask instructs the image list to draw an image, and draw the overlay mask on top of it.

MORE INFORMATION

To overlay images correctly using image lists, follow these steps:

1. Create a bitmap that will have the images you want to draw as well as the overlay images you want drawn on top of these images.

   For example, say you have a bitmap of four 16x16 images:

   - a green circle.
   - a red circle.
   - a panda.
   - a frog.

2. Create an image list out of the bitmap you've created in step 1 by using ImageList_LoadImage() as shown here:

```
hImageList = ImageList_LoadImage (hInst,
                                  "MyBitmap",
                                  16,
                                  4,
                                  RGB (255,0,0),
                                  IMAGE_BITMAP,
                                  0);
```

3. Decide which images you want to specify as overlay masks, and tag them as such by using the ImageList_SetOverlayImage() function. The following

code specifies the panda and the frog (with 0-based index, this comes out to image 2 and 3) as overlay masks 1 and 2.

NOTE: You can only specify up to four overlay masks per image list.

```
ImageList_SetOverlayImage (hImageList,
                           2,         // 0-based index to image list
                           1);        // 1-based index to overlay mask.


ImageList_SetOverlayImage (hImageList,
                           3,       // 0-based index to image list
                           2);      // 1-based index to overlay mask.
```

4. Draw the image. The following code draws the green circle (or image 0 in the example image list). Then it draws the panda (overlay image 1 in the example) on top of it.

```
ImageList_Draw (hImageList,
                0,     // 0-based index to imageList of image to draw
                hDC,   // handle to a DC
                16, 16 // (x,y) location to draw
                INDEXTOOVERLAYMASK (1));   // Overlay image #1
```

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrCtl

How to Override Full Drag

PSS ID Number: Q121541

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Windows NT version 3.5 introduces full drag, which allows you to see the entire window moving or resizing instead of seeing just an outline of the window moving or resizing. You can enable full drag by running the Desktop Control Panel applet and selecting the Full Drag checkbox.

When you resize a window with full drag enabled, the application will receive numerous messages indicating that the window is resizing. (You can verify this with Spy.) If this has undesirable effects on your application, you will need to override the full drag feature in your application.

When the moving or resizing starts, the application receives this message:

```
WM_ENTERSIZEMOVE (0231)
```

When the moving or resizing finishes, the application receives this message:

```
WM_EXITSIZEMOVE (0232)
```

The above messages act as a notification that the window is entering and exiting a sizing or moving operation. If you want, you can use these notifications to set a flag to prevent the program from handling a WM_PAINT message during the move or size operation to override full drag.

Additional reference words: 3.50 3.51 4.00 95
KBCategory: kbui kbtool
KBSubcategory: UsrWndw

How to Pass Large Memory Block Through Win32s Universal Thunk

PSS ID Number: Q126708

The information in this article applies to:

- Microsoft Win32s version 1.2

SUMMARY

=======

You can pass a memory address to a thunk routine. The pointer address is translated via the universal thunk (UT). However, the translated pointer is only guaranteed for 32K. This article describes ways to pass a larger memory block through the universal thunk.

MORE INFORMATION

================

GlobalAlloc()

You can call GlobalAlloc() to allocate a larger memory block on the 32-bit side of the thunk, copy the data into this memory block, send the handle to the 16-bit side, and lock the handle on the 16-bit side with GlobalLock().

VirtualAlloc()

If you allocate the memory using VirtualAlloc(), it will be aligned on a 64K boundary, so that you can address the entire memory block. HeapAlloc() allocates large memory blocks using VirtualAlloc() as well. NOTE: You are still limited to 64K of memory, due to the selector tiling.

Allocate a selector

To use this method, get the 32-bit offset used by the Win32-based application and the selector base for the data selector returned by GetThreadSelectorEntry(), then calculate the linear address of the memory block. With this linear address, you can use AllocSelector(), SetSelectorBase(), and SetSelectorLimit() to access the memory block from the 16-bit side of the thunk.

NOTE: Sparse memory will cause problems in the general case. Make sure that the memory range has been not only reserved, but also committed.

Additional reference words: 1.20

KBCategory: kbprg

KBSubcategory: W32s

How To Pass Numbers to a Named Range in Excel through DDE

PSS ID Number: Q45714

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

To send an array to Excel via DDE, you must send the array in TEXT, CSV, or BIFF clipboard format.

For example, if you want to send three numbers for one ROW in CSV format, use 1,2,3 (where each number is separated by a comma). If you want to place three numbers in one COLUMN in CSV format, place the CR/LF (Od 0a) (carriage return/line feed) characters after each number in the set.

If you would like to send the numbers via the TEXT format separated into columns, place the CR/LF characters between each number in the set. To organize the numbers into one row, place a TAB (09) character between each number.

In order to send an array of data into a named range, use the following steps:

1. Highlight the appropriate cells in Excel
2. Set up your typical hot link from Excel, for example:

   =Service|Topic!Item

3. Instead of hitting Enter after typing the above, hit Ctrl+Shift+Enter. This will cause your data to come in as an array, rather than as a single item.

   NOTE: To do the reverse of this, that is, for a client application to POKE data to Excel, the client will have to specify an item name of say, "R1C1:R1C2" to poke an array of data to the range R1C1..R1C2.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

How to Perform Auto Repeat as Media Player Does

PSS ID Number: Q124185

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0
- Microsoft Video for Windows Development Kit (DK) version 1.1

The Media Player (MPLAYER.EXE) included with Microsoft Windows and Microsoft Windows NT (MPLAY32.EXE) provides an auto-repeat option that automatically repeats the playback of a multimedia file. You can incorporate this functionality into your application on Digital Video devices by using an extension to the standard Media Control Interface (MCI) commands as follows:

- When calling the mciSendString() function, add the word "repeat" to the play command, as in this example:

```
mciSendString("play mov notify repeat", NULL, 0, hWnd);
```

- When calling the mciSendCommand() function, set the play flag MCI_DGV_PLAY_REPEAT. For example, to add auto repeat functionality to the MOVPLAY sample included with the Video for Windows DK, add the following line to the playMovie() function in MOVPLAY1.C, right before the mciSendCommand() function call:

```
dwFlags |= MCI_DGV_PLAY_REPEAT;
```

"Digital Video Command Set for the Media Control Interface" documents the Digital Video MCI extensions. It is available on the Microsoft Developer Network (MSDN) CD. Look for it in the Specifications section of the CD contents, under "Digital Video MCI Specification." You can also search the CD using the word MCI_DGV_PLAY_REPEAT for more information about that flag.

Additional reference words: 3.10 3.50 4.00 95 Video for Windows MCIAVI MCI_PLAY AVI loop continuous play
KBCategory: kbmm kbprg kbdocerr
KBSubcategory: MMVideo

How to Port a 16-bit DLL to a Win32 DLL

PSS ID Number: Q125688

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

There are several significant differences between Win16 DLLs and Win32 DLLs. These differences require more than just a simple recompilation to turn your Win16 DLL into a Win32 DLL. In this article we will show you how to port your Win16 DLL to a Win32 DLL.

MORE INFORMATION
================

The first major difference is that the Win32 DLL entry point is called with every process attach and detach. Secondly, you must account for the fact that processes can be multithreaded and as such your DLL entry point will be called with thread attach and detach messages. You need to ensure that your DLL is "thread-safe" by using multithreaded libraries and mutual exclusion for functions in your DLL that would otherwise cause data corruption when preempted and reentered. This requires that you use Win32 synchronization methods to guard critical resources. Finally, each Win32 process gets its own copy of the Win32 DLL's data.

Step One for Porting Your DLL

The first step in porting a DLL from 16-bit Windows to 32-bit Windows is moving code from your LibMain (or LibEntry) and _WEP (or WEP) to the new DLL initialization function. The new DLL initialization function is called DllMain. You might code your DllMain like this:

```
BOOL WINAPI DllMain (HANDLE hModule, DWORD fdwReason, LPVOID lpReserved)
{
   switch (fdwReason)
   {
      case DLL_PROCESS_ATTACH:
         /* Code from LibMain inserted here.  Return TRUE to keep the
            DLL loaded or return FALSE to fail loading the DLL.

            You may have to modify the code in your original LibMain to
            account for the fact that it may be called more than once.
            You will get one DLL_PROCESS_ATTACH for each process that
            loads the DLL. This is different from LibMain which gets
            called only once when the DLL is loaded. The only time this
            is critical is when you are using shared data sections.
```

```
            If you are using shared data sections for statically
            allocated data, you will need to be careful to initialize it
            only once. Check your code carefully.

            Certain one-time initializations may now need to be done for
            each process that attaches. You may also not need code from
            your original LibMain because the operating system may now
            be doing it for you.
         */
         break;

      case DLL_THREAD_ATTACH:
         /* Called each time a thread is created in a process that has
            already loaded (attached to) this DLL. Does not get called
            for each thread that exists in the process before it loaded
            the DLL.

            Do thread-specific initialization here.
         */
         break;

      case DLL_THREAD_DETACH:
         /* Same as above, but called when a thread in the process
            exits.

            Do thread-specific cleanup here.
         */
         break;

      case DLL_PROCESS_DETACH:
         /* Code from _WEP inserted here.  This code may (like the
            LibMain) not be necessary.  Check to make certain that the
            operating system is not doing it for you.
         */
         break;
   }

   /* The return value is only used for DLL_PROCESS_ATTACH; all other
      conditions are ignored.  */
   return TRUE;   // successful DLL_PROCESS_ATTACH
}
```

DllMain Called with Flags

There are several conditions where DllMain is called with the DLL_PROCESS_ATTACH, DLL_PROCESS_DETACH, DLL_THREAD_ATTACH, or DLL_THREAD_DETACH flags.

The DLL_PROCESS_ATTACH flag is sent when a DLL is loaded into the address space of a process. This occurs in both situations where the DLL is loaded with LoadLibrary, or implicitly during application load. When the DLL is implicitly loaded, DllMain is executed with DLL_PROCESS_ATTACH before the processes enter WinMain. When the DLL is explicitly loaded, DllMain is executed with DLL_PROCESS_ATTACH before LoadLibrary returns.

The DLL_PROCESS_DETACH flag is sent when a process cleanly unloads the DLL from its address space. This occurs during a call to FreeLibrary, or if the DLL is implicitly loaded, a clean process exit. When a DLL is detaching from a process, the individual threads of the process do not call the DLL_THREAD_DETACH flag.

The DLL_THREAD_ATTACH flag is sent when a new thread is being created in a process already attached to the DLL. Threads in existence before the process attached to a DLL will not send the DLL_THREAD_ATTACH flag. The first thread to attach to the DLL does not send the DLL_THREAD_ATTACH flag; it sends the DLL_PROCESS_ATTACH flag instead.

The DLL_THREAD_DETACH flag is sent when a thread is exiting cleanly. There is a situation when DllMain may be called when the thread did not first send the DLL_THREAD_ATTACH flag. This can happen if there are other threads still running and the original thread exits cleanly. The thread originally called DllMain with the DLL_PROCESS_ATTACH flag and later calls DllMain with the DLL_THREAD_DETACH flag. You may also have DllMain being called with DLL_THREAD_DETACH if a thread exits but was running in the process before the call to LoadLibrary.

Situations Where DllMain is Not Called or Is Bypassed

DllMain may not be called at all in dire situations where a thread or process was killed by a call to TerminateThread or TerminateProcess. These functions bypass calling DllMain. They are recommended only as a last resort. Data owned by the thread or process is at risk of loss because the process or thread could not shut itself down properly.

DllMain may be bypassed intentionally by a process if it calls DisableThreadLibraryCalls. This function (available with Windows 95 and Windows NT versions 3.5 and later) disables all DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications for a DLL. This enables a process to reduce its code size and working set. For more information on this function, see the SDK documentation on DisableThreadLibraryCalls.

Step Two for Porting Your DLL

The second step of porting your DLL involves changing functions that were declared with __export and included in the module definition (.DEF) file EXPORTS section. The proper export declaration for the Microsoft Visual C++ 32-bit compiler and linker is __declspec(dllexport). This declaration should be used when prototyping and declaring functions. You do not have to explicitly declare an exported function in the DEF file for the proper LIB and EXP files to be created; __declspec(dllexport) will do this for you. Here's an example of an exported function:

```
// prototype in DLL is necessary
__declspec(dllexport) DWORD WINAPI DLLFunc1(LPSTR);

// function
__declspec(dllexport) DWORD WINAPI DLLFunc1(LPSTR lpszIn)
```

```
{
   DWORD dwRes;

   /* DLL function logic */

   return dwRes;
}
```

To include the function in an application, prototype the above function in the application with the __declspec(dllimport) modifier. __declspec(dllimport) is not necessary, but does improve the speed of your code that implements the function call. Here's an example:

```
   __declspec(dllimport) DWORD WINAPI DLLFunc1(LPSTR);
```

Then, link the DLL's import library (.LIB) file with the application makefile or project.

Some sections of your DEF file will be ignored by the 32-bit linker because of architectural differences between Win16 and Win32. You may still use the EXPORTS section of your DEF if you wish to include ordinals for exported functions, or to rename exported functions. See your linker documentation for more information about what is acceptable in a DEF file for a Win32 DLL. Users of other 32-bit compilers and linkers will have to refer to their documentation for more information on exporting functions.

Applications that link to your DLL may be multithreaded. This possibility means that you should always build your Win32 DLL as multithreaded to support preemption and reentrancy. If you use runtime library functions in your DLL, they may be preempted and reentered. That would cause problems for a normal runtime library. If you use C runtime or some other runtime library, you should use a multithreaded version of the runtime library. Microsoft Visual C++ users should link with the /MT option and include LIBCMT.LIB. This will include the multithreaded C runtime library. Optionally, you can select the Multithreaded Run-time Library option in the project settings in the Visual Workbench. If you are using another runtime library and cannot get a multithreaded version of the library, you must protect calls to the library from reentrancy using a mutex or critical section synchronization object. Information later in this article discusses this issue.

Other Design Issues You Should Consider

One of the most significant changes to DLLs in 32-bit Windows operating systems is that each process executes in its own private address space. This means that a DLL cannot directly share dynamically-allocated memory between processes. Addresses are 32-bit offsets in a process's address space. Passing them between processes is possible, but won't work as in Win16 because each process has its own address space. In another process, this pointer may address unknown data, or invalid memory space.

"DS != SS" issues common to 16-bit DLLs no longer apply. A Win32 process executes in its own private address space and there is no segmentation of this address space. In a Win32 DLL, all functions are called using the

calling thread's stack and all pointers are 32-bit linear addresses. In a Win32 process or DLL, a FAR pointer is the same as a near pointer. In other words, a pointer is just a pointer.

If you must share data, you can specify certain global variables to be shared among processes by using a shared data section in the DLL. For more information on shared data sections, please search the Microsoft Knowledge Base using the following words:

   #pragma data_seg

or:

   sections share dll

You can also use a file-mapping object to share memory by sharing the system page file. This will allow two different processes to share dynamic memory. For more information on file mapping objects, please search the Microsoft Knowledge Base using these words:

   shared memory

Another significant change in the behavior of Win32 DLLs from Win16 DLLs is the inclusion of synchronization. The Win32 API provides synchronization objects that allow the programmer to implement correct synchronization. You should be aware that your 32-bit DLL may be preempted and called again from a different thread in the process. For example, if a thread executing a DLL function accesses global data and is preempted and another function modifies the same data, the original thread will resume but will be using modified data. You'll need to use synchronization objects to resolve this situation.

Your Win32 DLL may also be preempted by a thread in a different process. This situation becomes important if the functions use shared data sections or file mappings. You will need to examine the functions in a DLL to determine if this will cause problems. If so, you will have to control access to data or sections of code that are sensitive to this problem. Mutexes and critical sections are well suited for DLL synchronization. For more information on synchronization, please search the Microsoft Knowledge Base using these words:

   synchronization objects

For additional information, please search the Microsoft Knowledge Base using these words:

   Win32 DLL

Additional reference words: 4.00 LibEntry LibMain Port WEP _WEP Win32
KBCategory: kbprg
KBSubcategory: BseDll

How to Print a Document

PSS ID Number: Q139652

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API)
  included with:

   - Microsoft Windows NT versions 3.5 and 3.51
   - Microsoft Windows 95 version 4.0

SUMMARY
=======

This article describes each of the seven steps required to print a document to a printer in Windows programming.

MORE INFORMATION
================

1. Obtain a Printer Device Context

To draw graphics or text on the printer device, an application needs to obtain a printer device context. The PrintDlg() function can be used to obtain the printer DC. PrintDlg() can display a Print Dialog box to allow the user to select a printer, or it can return information about the default printer. In addition to other information about the printer, PrintDlg() will return a printer device context in the PRINTDLG structure when PD_RETURNDC is specified as one of the flags. This device context matches the selections the user made in the dialog box. The GetPrinterDC function in the sample code at the end of this article illustrates the use of PrintDlg() to obtain a printer DC.

If you want to create a printer DC without displaying the Print dialog box, then you need to specify PD_RETURNDEFAULT | PD_RETURNDC flags as shown in the sample code in this article. The PD_RETURNDEFAULT flag is used to retrieve information about the default printer without displaying the Print dialog box. PD_RETURNDC flag is used to direct PrintDlg to automatically create a device or information context for the printer.

2. Set Up the Abort Function

An application must use the SetAbortProc function to set the application-defined abort function that allows a print job to be canceled during spooling. In the AbortProc function, the abort procedure can check the error code to see if an error occured while printing. The error code is zero if no error has occured.

3. Use StartDoc() to Start the Print Job

The StartDoc function starts a print job. A DOCINFO structure is initialized and passed to the StartDoc function. It is a good idea to initialize the DOCINFO structure by filling it with zeros. For more information, please see the following article in the Microsoft Knowledge Base:

```
   ARTICLE-ID: Q135119
   TITLE     : PRB: StartDoc() Fails with Non-Zeroed DOCINFO
```

The InitDocStruct function illustrated later in this article performs this initialization.

4. Call StartPage()

The StartPage function prepares the printer driver to accept data. For example:

```
  StartPage( hDC );
```

5. Draw on the Device Context

Draw graphics or text on the printer device. For example, DrawStuff() illustrates how to draw text on the printer DC.

6. Call EndPage()

The EndPage function informs the device that the application has finished writing to a page. This function is typically used to direct the device driver to advance to a new page. To print multiple pages, Steps 4, 5, and 6 must be used for every page of the document as in this example:

```
for( i = START_PAGE; i <= END_PAGE; i++)
  {
    StartPage();
    DrawStuff();
    EndPage();
  }
```

7. Call EndDoc()

The EndDoc function ends a print job. For additional information on this topic, please refer to the Win32 SDK documentation Overviews section.

Sample Code

The following PrintStuff() function illustrates the printing process.

```
/*============================================*/
/* Sample code :  Typical printing process      */
/* ============================================*/
```

```
void PrintStuff( HWND hWndParent )
{
    HDC        hDC;
    DOCINFO    di;

    // Need a printer DC to print to
    hDC = GetPrinterDC();

    // Did you get a good DC?
    if( !hdc)
    {
        MessageBox(NULL, "Error creating DC", "Error",
                                    MB_APPLMODAL | MB_OK );
        return;
    }

    // You always have to use an AbortProc()
    if( SetAbortProc( hDC, AbortProc ) == SP_ERROR )
    {
        MessageBox( NULL, "Error setting up AbortProc",
                                    "Error", MB_APPLMODAL | MB_OK);
        return;
    }

    // Init the DOCINFO and start the document
    InitDocStruct( &di, "MyDoc");
    StartDoc( hDC, &di );

    // Print one page
    StartPage( hDC );
    DrawStuff( hDC );
    EndPage( hDC );

    // Indicate end of document
    EndDoc( hDC );

    // Clean up
    DeleteDC( hDC );
}

/*================================*/
/* Obtain printer device context */
/* ================================*/
HDC GetPrinterDC(void)
{
    PRINTDLG pdlg;

    // Initialize the PRINTDLG structure
    memset( &pdlg, 0, sizeof( PRINTDLG ) );
    pdlg.lStructSize = sizeof( PRINTDLG );
    // Set the flag to return printer DC
    pdlg.Flags = PD_RETURNDEFAULT | PD_RETURNDC;

    // Invoke the printer dialog box
```

```
    PrintDlg( &pdlg );
    // hDC member of the PRINTDLG structure contains
    // the printer DC
    return pdlg.hDC;
}

/*================================*/
/* The Abort Procudure            */
/* ================================*/
BOOL CALLBACK AbortProc( HDC hDC, int Error )
{
    MSG   msg;
    while( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    return TRUE;
}

/*================================*/
/* Initialize DOCINFO structure  */
/* ================================*/
void InitDocStruct( DOCINFO* di, char* docname)
{
    // Always zero it before using it
    memset( di, 0, sizeof( DOCINFO ) );
    // Fill in the required members
    di->cbSize = sizeof( DOCINFO );
    di->lpszDocName = docname;
}

/*================================*/
/* Drawing on the DC              */
/* ================================*/
void DrawStuff( HDC hdc)
{
    // This is the function that does draws on a given DC
    // Here, you are printing text
    TextOut(hdc, 0,0, "Test Printing", lstrlen( "Test Printing" ) );
}
```

Additional reference words: 3.50 4.00 Windows 95
KBCategory: kbprint kbcode kbhowto
KBSubcategory: GdiPrn

How to Program Keyboard Interface for Owner-Draw Menus

PSS ID Number: Q121623

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

You can implement the keyboard interface for owner-draw menus, which allow a user to access a menu by typing a menu mnemonic, by processing the WM_MENUCHAR message.

MORE INFORMATION

Menus other than owner-draw menus can specify a menu mnemonic by inserting an underscore next to a character in the menu string so that the user can select the menu by typing ALT+<menu mnemonic character>. But in owner-draw menus, you cannot specify a menu mnemonic in this manner. Instead, you must process the WM_MENUCHAR message to provide owner-draw menus with menu mnemnoics.

WM_MENUCHAR is sent when the user types a menu mnemonic that does not match any of the predefined mnemonics of the current menu. wParam specifies the ASCII character that corresponds to the key the user pressed together with the ALT key. The low-order word of lParam specifies the type of the selected menu and contains:

- MF_POPUP if the cuurent menu is a popup menu.
- MF_SYSMENU if the menu is the system menu.

The high-order word of lParam contains the menu handle of the current menu. The window with the owner-draw menus can process WM_MENUCHAR as follows:

```
   case WM_MENUCHAR:
      nIndex = Determine index of menu item to be selected from
               character that was typed and handle of the current
               menu.
      return MAKELRESULT(nIndex, 2);
```

The 2 in the high-order word of the return value informs Windows that the low-order word of the return value contains the zero-based index of the menu item to be selected by Windows.

Windows 95 defines four new constants that correspond to the possible return values from the WM_MENUCHAR message:

Constant	Value	Meaning
MNC_IGNORE	0	Informs Windows that it should discard the character the user pressed and create a short beep on the system speaker.
MNC_CLOSE	1	Informs Windows that it should close the active menu.
MNC_EXECUTE	2	Informs Windows that it should choose the item specified in the low-order word of the return value. The owner window receives a WM_COMMAND message.
MNC_SELECT	3	Informs Windows that it should select the item specified in the low-order word of the return value.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbui
KBSubcategory: UsrMen

How to Reference Colors in a DIB Section by Index

PSS ID Number: Q138256

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, 4.0

SUMMARY

When using the GDI with DIB sections (HBITMAPs returned by a call to CreateDIBSection()), you may need a way to reference the colors in the color table of the DIB section by index rather than by RGB value. This article shows you how.

MORE INFORMATION

To reference a color table value by index, use the DIBINDEX macro defined in Mmsystem.h. It allows you to index the colors in the color table of a DIB section in a manner similar to the way PALETTEINDEX indexes a color in a logical palette. In other words, you can create COLORREFs that reference a DIB section's color table rather than the logical palette in the DC that the DIB section is selected into. Because DIBINDEX references values in a color table, it only works on DCs where you have a DIB section selected.

The DIBINDEX macro accepts an index to a color table entry and returns a color table specifier consisting of a 32-bit COLORREF value that specifies the color associated with the given index. An application using a display context with a DIB section selected into it can pass this specifier, instead of an explicit red, green, blue (RGB) value, to GDI functions that expect a color. This allows the function to use the color at the specified color table index.

```
LONG DIBINDEX(
    WORD  wColorTableIndex    // index to color table entry
   );
```

Parameters

wColorTableIndex - Specifies an index to the palette entry containing the color to be used for a graphics operation.

Return Value

The return value is a color table index specifier.

Remarks

DIBINDEX is defined as:

```
#define DIBINDEX(n)     MAKELONG((n),0x10FF)
```

NOTE: DIBINDEX also works with 16-bit WinGBitmaps and WinGDCs.

Additional reference words: 3.10 3.50 4.00 Windows 95 COLORTABLE HBITMAP
BRUSH COLORREF PEN DIBSECTION
KBCategory: kbprg kbhowto kbcode
KBSubcategory: W32

How to Remotely Debug a Win32s Application

PSS ID Number: Q133061

The information in this article applies to:

- Microsoft Win32s, version 1.25

This article describes how to remotely debug a Win32s application. Before you can start remote debugging your application, you must make sure you have a valid connection between the two machines you are using. One way you can test this functionality is by using the terminal emulator found in the Accessories group. If you can send text over the serial line then you have a valid connection. If you do not have a valid connection, make sure the correct COM ports are selected, and verify the baud speed is the same on both machines.

NOTE: You must use a null modem cable to create a valid connection between the two machines.

The following steps can be used to set up remote debugging to work over a serial line:

1. Make sure Win32s is loaded on the Windows 3.11 machine.

2. Make sure you have the following files in your Window 3.11 System subdirectory:

   ```
   Wdbg32s.exe
   Dm32s.dll
   Tlser32s.dll
   ```

   These files can be placed in one subdirectory. You can install these files by using the Microsoft Windows version 3.11 Software Development Kit (SDK), or by copying them from your Windows NT machine if you have installed Windbg.exe from the Windows SDK.

3. Place the executable and debug information that you want to debug on your Windows 3.11 machine. Note the path to the executable because this information is used in step 4.

4. On your Windows NT machine, click Open on the Program menu from the Windbg.exe program. Click New in the dialog box, and DO NOT BROWSE FOR THE FILE. In the Edit box, type the path that you noted in step 3. Windbg.exe needs the path of your remote executable because Windbg.exe looks for a remote executable.

5. Select the appropriate serial connection on both machines. For example, ser96 means use COM1 at 9600 baud. You can modify this information if you need to change these parameters by clicking the Change button located at the bottom right of the dialog box. Refer to either the Windebug.exe or Windbg.exe help file for more information on modifying the default COM port and baud rates.

6. On your Windows NT machine, change the default transport DLL, which is most likely Tlserr.dll, to Tlser32.dll. Use this DLL (Tlser32.dll) when you are remotely debugging a Win32s application. This is done by clicking the Change button with the Transport Layer edit box highlighted. An edit box labeled PATH with the name of the DLL is displayed. Verify it contains Tlser32.dll. When executing Wdbg32s on your Windows 3.11 machine, make sure the transport DLL is Tlser32s.dll. You can select this DLL by clicking "Transport DLL..." in the Options menu of Winddbgrm.exe.

7. On the Windows 3.11 machine, make sure the connection settings are correct (COM port and baud speed settings should be the same that you entered in step 5 for the Windows NT machine.) Click Connect. The Connect option should be dimmed and the Disconnect menu option should be highlighted.

8. On your Windows NT machine, click Go on the Run menu.

You can now start remotely debugging your Win32s application.

Additional reference words: 3.50
KBCategory: kbtool
KBSubcategory: TlsDbgWin TlsDbg

How to Remove Win32s

PSS ID Number: Q120486

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, 1.15, 1.2, and 1.30A

SUMMARY
=======

To remove Win32s:

1. Remove the following line from the [386Enh] section in the SYSTEM.INI file:

   device=<WINDOWS>\<SYSTEM>\win32s\w32s.386

   where <WINDOWS> and <SYSTEM> are where the Windows and System directories are, respectively.

2. Modify the following line from the [BOOT] section in the SYSTEM.INI file:

   drivers=mmsystem.dll winmm16.dll

   to the following (remove winmm16.dll):

   drivers=mmsystem.dll

3. Delete the following files from the <WINDOWS>\<SYSTEM> subdirectory or from the SYSTEM directory in network installations:

   WINMM16.DLL
   W32SYS.DLL
   WIN32S16.DLL
   WIN32S.INI
   WINHLP32.EXE
   WINDOWS.HLP
   WINHLP32.HLP
   WINHLP32.CNT
   *
   * MSINTERNAL:
   * The following files are created by WINHLP32.EXE and they might exist
   * in the WINDOWS\SYSTEM directory:
   *
   *   winhlp32.fts
   *   winhlp32.ftg
   *   winhlp32.gid
   *
   * WINHLP32.GID file is a hidden file; you have to change its attribute
   * to not hidden before you can delete it. You may choose not to confuse
   * customers with these additional files, especially because the files
   * may not even exist. Per Boaz Feldbaum Oct. 8, 1995.

   Note: It may not be possible to delete W32SYS.DLL while Windows is
   running. This occurs if you ran some Win32 application. It is
   recommended that you remove Win32s while in MS-DOS, not from an MS-DOS
   box.

4. Delete all the files in the <WINDOWS>\<SYSTEM>\WIN32S subdirectory
   or the <SYSTEM>\WIN32S subdirectory in network installations.
   Then delete subdirectory itself.

5. Restart Windows.

NOTE: <WINDOWS> refers to the windows installation directory. On a networked Windows installation, the system directory may be located on a remote share. If you are only removing Win32s from your machine, then you do not need to remove the shared files (in <SYSTEM> and <SYSTEM>\WIN32S). Before removing these files from the network share, make sure that all users that use Win32s have removed the references to Win32s from the SYSTEM.INI file.

MORE INFORMATION
================

This information can be found in the Win32s Programmer's Reference help file.

Additional reference words: 1.00 1.10 1.15 1.20 1.30A
KBCategory: kbsetup kbprg
KBSubcategory: W32s

How to Restart the Windows Shell Programmatically

PSS ID Number: Q137572

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SUMMARY

Creating and updating shell extensions in Windows requires that the shell be restarted. This may be accomplished by having the user manually perform this task; however, in many cases, you may find it is better to do it programmatically.

MORE INFORMATION

To restart the shell programmatically, find the shell window, post it a quit message, and then call WinExec() with explorer.exe. For Example:

```
HWND hwndShell = FindWindow("Progman", NULL);
PostMessage(hwndShell, WM_QUIT, 0, 0L);
WinExec("Explorer.exe",SW_SHOW);
```

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbcode
KBSubcategory: UsrShell

How to Right-Justify Menu Items in Windows 95

PSS ID Number: Q125675

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

In Windows 95, right-justify (right-align) a menu item by using the MFT_RIGHTJUSTIFY type in MENUITEMINFOSTRUCTURE.

MORE INFORMATION

There is a new menu type in Windows 95, MFT_RIGHTJUSTIFY type, which you can use to right justify a menu item. The Windows version 3.1 method of prefixing the string with "\a" or "\b" will no longer work.

To right justify a menu item in Windows 95:

1. Get the menu handle of the original menu.

2. Get the original menu item information stored in the MENUITEMINFO structure.

3. Change the menu item type to include MFT_RIGHTJUSTIFY by or'ing the original value with MFT_RIGHTJUSTIFY.

4. Set the new menu item information.

For example, to create a right-justified menu item, add the following code to WM_CREATE:

```
   HMENU hMenu;
   MENUITEMINFO  mii;
   char szBuffer [80];

   hMenu = GetMenu (hwnd);

   // Get the original value of mii.fType first
   // and OR that with MFT_RIGHTJUSTIFY
   mii.cbSize = sizeof (MENUITEMINFO);
   mii.fMask = MIIM_TYPE;
   mii.dwTypeData= szBuffer;
   mii.cch   = sizeof (szBuffer);

   GetMenuItemInfo(hMenu, 1, TRUE, &mii);

   // OR in MFT_RIGHTJUSTIFY type
```

```
   mii.fMask = MIIM_TYPE;
   mii.fType  = mii.fType | MFT_RIGHTJUSTIFY;

   // Right justify the specified item and all those following it
   SetMenuItemInfo(hMenu, 1, TRUE,   &mii);

   return 0;
```

REFERENCES
==========

For additional information on right justifying menus in Windows 3.1, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q67063
TITLE   : Inserting Right Justified Text in a Menu in Windows

Additional reference words: 4.00 alignment align
KBCategory: kbui
KBSubcategory: UsrMen

How to Run Name Service API-Based RPC Apps in Windows 95

PSS ID Number: Q140016

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SUMMARY

A Win32 RPC application that calls name service RPC APIs cannot run without adding or configuring certain registry entries.

MORE INFORMATION

RPC Server and client applications relying on the services of Microsoft RPC locator service fail with errors 0x6BF(1727 - A renote procedure call failed and did not execute), and 0x6E2(1762 - The name service is unavailable) on calls to RpcNsBindingExport() and RpcNsBindingImportBegin() APIs, respectively.

This is due to the fact that Windows 95 presently does not have a Microsoft RPC locator component available. For these applications to work correctly, the user must modify the Windows 95 system registry, so that these RPC calls can be redirected to the a locator service running on a Windows NT server or a Windows NT workstation.

Adding the following registry keys in the specified path solves this problem:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\NameService\DefaultSyntax
REG_SZ "3"

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\NameService\Endpoint
REG_SZ "\pipe\locator"

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\NameService\NetworkAddress
REG_SZ "\\NTSERVER"

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\NameService\Protocol
REG_SZ "ncacn_np"

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Rpc\NameService\ServerNetworkAddress
REG_SZ "\\NTSERVER"

Here, NTSERVER is the computer name of the Windows NT server or workstation where the RPC locator service is already running. A Microsoft locator component is not currently available for Windows 95 because of the fact that the Microsoft locator uses named pipes to communicate, and Windows 95 does not support server-side named pipes.

Additional reference words: 4.00

KBCategory: kbprg kbnetwork
KBSubCategory: NtwkRpc

How to Search for Win32 SDK Articles by KB Subcategory

PSS ID Number: Q115696

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

SUMMARY

The Microsoft Knowledge Base (KB) is categorized by using keywords. This article lists the category and subcategory keywords specific to articles in the Microsoft Win32 Software Development Kit (SDK) for Windows NT and Windows 95 Collection.

MORE INFORMATION

Microsoft Win32 SDK KBSubcategory Major and Minor Keywords

Each KB article in the Win32 SDK collection may contain one or more product-specific subcategory keywords (called KBSubcategory keywords) that places the article in an appropriate Win32 SDK category. Each KBSubcategory word is composed of the concatenation of a major topic keyword and a minor topic keyword. For example, you can find all the BASE DLL articles by using BseDll as the keyword when you query the Microsoft Knowledge Base.

Use the asterisk (*) wildcard to find articles that fall into the general categories or into an intermediate subcategory. For example, to find all the articles that apply to GDI issues, query on Gdi*. An article usually has only one subcategory keyword, but it may have more.

Here are the topics and the corresponding KBSubcategory keywords. The minor topics are in the indented list under each major topic.

Major Topic / Minor Topic	KBSubcategory Keyword
BASE (Bse)	
Base Misc	BseMisc
Comm APIs	BseCommapi
Console	BseCon
Cryptography	BseCrypt
DLLs	BseDll
Error Debug	BseErrdebug
Exception Handling	BseExcept
Disk/File I/O	BseDskFileio
File I/O	BseFileio
Floating Point	BseFltpt
IPC	BseIpc

	Memory Management	BseMm
	Procedure Threads	BseProcThrd
	Security	BseSecurity
	Service	BseService
	Synchronization	BseSync
	Registry	BseRegistry
	Object Store	BseObjStore
	Notifications	BseNotify
	Thunks	BseThunks
	Remote SPI	BseRemoteAPI
GDI (Gdi)		
	Bitmaps	GdiBmp
	Brushes/Pens	GdiPnbr
	Cursors/Icons	GdiCurico
	DCs	GdiDc
	Display	GdiDisplay
	Drawing	GdiDrw
	Fonts	GdiFnt
	GDI Misc	GdiMisc
	Metafiles	GdiMeta
	OpenGL	GdiOpenGL
	Palettes/Colors	GdiPal
	Printing	GdiPrn
	Screen Saver	GdiScrsav
	TrueType	GdiTt
MAPI		
	Address Book Provider	MAPIIAB
	Client Extensibility	EXCHEXT
	EDKAPI	EDKAPI
	Extended MAPI Client	EMAPI
	MAPI Forms	MAPIFORM
	OLE Messaging	OLEMSG
	Simple MAPI CMC	SMAPICMC
	Store Provider	MAPIIMS
	Transport Provider	MAPIIXP
PMAIL		
	Transport Provider	PMAILTrans
PIM		
	Address Book	PIMAddr
MISC		
	International Issues	WIntlDev
	Setup/Install	Setins
	Subsystems	SubSys
MULTIMEDIA		
	ActiveMovie	MMActiveMovie
	CD-ROM	MMCDROM
	DirectInput	MMDirectInput
	DirectSound	MMDirectSound
	Joystick	MMJoy

MediaView	MMMediaView
Midi	MMMidi
Miscellaneous	MMMisc
Multimedia Timers	MMTimer
MM File I/O Services	MMIO
Sound Card Mixer Control	MMMixer
Speech SDK	MMSpeech
Video for Windows (VFW)	MMVideo
Waveform Audio	MMWave
NETWORKING (Ntwk)	
LAN Manager APIs	NtwkLmapi
NetBIOS	NtwkNetbios
Networking Misc	NtwkMisc
RAS APIs	NtwkRAS
RPC	NtwkRpc
SNMP	NtwkSnmp
TC/PIP	NtwkTcpip
WNet APIs	NtwkWinnet
Windows Sockets	NtwkWinsock
PEN	Wpen*
Video Drivers	WpenVideoDrv
Pen Drivers	WpenTbltDrv
OAK	WpenOemOak
Recognizer	WpenRcgnzr
Dictionary	WpenDict
Visual Basic	WpenVB
Training	WpenTrain
RC	WpenRc
RCRESULT	WpenRcResult
SYG	WpenSyg
SYV	WpenSyv
SYC	WpenSyc
SYE	WpenSye
Ink	WpenInk
Setup	WpenSetup
Pen applets	WpenApps
control panel apps	WpenCpl
PEN UI	WpenPenUI
Gestures	WpenGestures
Keyboard	WpenKeyboard
Pen Misc	WpenMisc
TAPI	
Telephony API	Tapi
TOOLS (Tls)	
Dialog Editor	TlsDlg
Font Editor	TlsFnt
Headers	TlsHdr
Help Compiler	TlsHlp
Image Editor	TlsImg
MEP	TlsMep
Misc	TlsMisc

MS Setup	TlsMss
Porting Tool	TlsPort
Profiler	TlsPrf
Resource Compiler	TlsRc
Spy	TlsSpy
WinDbg	TlsWindbg
Pegasus Loader	TlsLoad
HTML Help	TlsHelp

USER (Usr)

Classes	UsrCls
Clipboard	UsrClp
Common Dialogs	UsrCmnDlg
Controls	UsrCtl
DDE	UsrDde
Dialog/Message Boxes	UsrDlgs
Drag & Drop	UsrDnd
Extension Libraries	UsrExt
Hooks	UsrHks
Input-Mouse/Keyboard	UsrInp
Localization/Intntl.	UsrLoc
MDI	UsrMdi
Menus/Accelerators	UsrMen
Network DDE	UsrNetDde
New Shell issues	UsrShell
NLS	UsrNls
Owner Draw	UsrOwn
Resources	UsrRes
Strict	UsrStrict
User Misc	UsrMisc
Window Manager	UsrWndw

WIN32S

Win32s	W32s
Win32s Thunk	W32sThunk
Win32s Setup Issues	W32sSetup

Product-Specific Keywords

You can use the KBSubcategory keywords to organize Win32 SDK articles or to search for specific groups of Win32 SDK articles. For information about KBSubcategory keywords for other Microsoft developer products, please query the Microsoft Knowledge Base using these keywords:

```
dskbguide and kbkeyword
```

KB-Wide Keywords

Each article in the Win32 SDK collection also contains at least one generic, KB-wide category keyword (called a KBCategory keyword). The KBCategory keywords are standard throughout the Microsoft Knowledge Base, appearing in all KB articles, regardless of product. You can use the KBCategory keywords to organize all KB articles or to search for articles

across several Microsoft products. For more information about these KBCategory keywords, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q94671
   TITLE   : Categories and Keywords for All Knowledge Base Articles

Additional reference words: 3.10 3.50 4.00
KBCategory: kbref kbkeyword
KBSubcatgory: GenSDK

How to Secure Performance Data in Windows NT

PSS ID Number: Q146906

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.51

SUMMARY

Windows NT provides access to a variety of performance data that collectively represents the state of the computer. This performance data is stored in the registry key HKEY_PERFORMANCE_DATA. The default configuration of Windows NT gives everyone the ability to query this performance data, including remote users.

In some environments, you may want to restrict access to this performance data because some performance data may be considered sensitive. An example of potentially sensitive performance data is the list of running processes in the system. This article describes how to regulate access to this performance data programmatically by using the Win32 API.

MORE INFORMATION

The security on the following registry key dictates which users or groups can gain access to the performance data:

```
HKEY_LOCAL_MACHINE\
 SOFTWARE\
 Microsoft\
 Windows NT\
 CurrentVersion\
 Perflib
```

In order for users to query performance data, they must have KEY_READ access to the above registry key. An example of reasonable security on the performance data would be to grant Administrators KEY_ALL_ACCESS access and Interactive (users logged onto the workstation interactively) KEY_READ access. This particular configuration would prevent non-administrator remote users from querying performance data.

Note that this operation can be performed by using the registry editor utility (Regedt32.exe).

Sample Code

```
/*
 This sample illustrates how to regulate access to the performance data
 provided by the registry key HKEY_PERFORMANCE_DATA.
```

```
 The security on the following registry key dictates which users or groups
 can gain access to the performance data:

 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib

 This sample opens the registry key for WRITE_DAC access, which allows
 for a new Dacl to be applied to the registry key.

 A Dacl is then built, which grants the following users access:

 Administrators are granted full control to allow for future updates to the
 security on the key and to allow for querying performance data.

 Interactively logged on users, through the well-known Interactive Sid,
 are granted KEY_READ access, which allows for querying performance
 data.

 The new Dacl is then applied to the registry key using the
 RegSetKeySecurity() Win32 API.

 This sample relies on the import library Advapi32.lib.
 Note that not all errors will cause an information message to be
 displayed.

 */

#include <windows.h>
#include <stdio.h>

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

int
__cdecl
main(
    void
    )
{
    SID_IDENTIFIER_AUTHORITY sia = SECURITY_NT_AUTHORITY;
    PSID pInteractiveSid = NULL;
    PSID pAdministratorsSid = NULL;
    SECURITY_DESCRIPTOR sd;
    PACL pDacl = NULL;
    DWORD dwAclSize;
    HKEY hKey;
    LONG lRetCode;
    BOOL bSuccess = FALSE; // assume this function fails

    //
    // open the performance key for WRITE_DAC access
    //
    lRetCode = RegOpenKeyEx(
        HKEY_LOCAL_MACHINE,
       TEXT("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Perflib"),
```

```
    0,
    WRITE_DAC,
    &hKey
    );

if(lRetCode != ERROR_SUCCESS) {
    fprintf(stderr, "RegOpenKeyEx error! (rc=%lu)\n", lRetCode);
    return RTN_ERROR;
}

//
// prepare a Sid representing any Interactively logged-on user
//
if(!AllocateAndInitializeSid(
    &sia,
    1,
    SECURITY_INTERACTIVE_RID,
    0, 0, 0, 0, 0, 0, 0,
    &pInteractiveSid
    )) goto cleanup;

//
// preprate a Sid representing the well-known admin group
//
if(!AllocateAndInitializeSid(
    &sia,
    2,
    SECURITY_BUILTIN_DOMAIN_RID,
    DOMAIN_ALIAS_RID_ADMINS,
    0, 0, 0, 0, 0, 0,
    &pAdministratorsSid
    )) goto cleanup;

//
// compute size of new acl
//
dwAclSize = sizeof(ACL) +
    2 * ( sizeof(ACCESS_ALLOWED_ACE) - sizeof(DWORD) ) +
    GetLengthSid(pInteractiveSid) +
    GetLengthSid(pAdministratorsSid) ;

//
// allocate storage for Acl
//
pDacl = (PACL)HeapAlloc(GetProcessHeap(), 0, dwAclSize);
if(pDacl == NULL) goto cleanup;

if(!InitializeAcl(pDacl, dwAclSize, ACL_REVISION))
    goto cleanup;

//
// grant the Interactive Sid KEY_READ access to the perf key
//
if(!AddAccessAllowedAce(
    pDacl,
```

```
        ACL_REVISION,
        KEY_READ,
        pInteractiveSid
        )) goto cleanup;

    //
    // grant the Administrators Sid GENERIC_ALL access to the perf key
    //
    if(!AddAccessAllowedAce(
        pDacl,
        ACL_REVISION,
        KEY_ALL_ACCESS,
        pAdministratorsSid
        )) goto cleanup;

    if(!InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION))
        goto cleanup;

    if(!SetSecurityDescriptorDacl(&sd, TRUE, pDacl, FALSE)) {
        fprintf(stderr, "SetSecurityDescriptorDacl error! (rc=%lu)\n",
            GetLastError());
        goto cleanup;
    }

    //
    // apply the security descriptor to the registry key
    //
    lRetCode = RegSetKeySecurity(
        hKey,
        (SECURITY_INFORMATION)DACL_SECURITY_INFORMATION,
        &sd
        );

    if(lRetCode != ERROR_SUCCESS) {
        fprintf(stderr, "RegSetKeySecurity error! (rc=%lu)\n",
            lRetCode);
        goto cleanup;
    }

    bSuccess = TRUE; // indicate success

cleanup:

    RegCloseKey(hKey);
    RegCloseKey(HKEY_LOCAL_MACHINE);

    //
    // free allocated resources
    //
    if(pDacl != NULL)
        HeapFree(GetProcessHeap(), 0, pDacl);

    if(pInteractiveSid != NULL)
        FreeSid(pInteractiveSid);
```

```
    if(pAdministratorsSid != NULL)
        FreeSid(pAdministratorsSid);

    if(!bSuccess) return RTN_ERROR;

    return RTN_OK;
}
```

Additional reference words: perfmon performance
KBCategory: kbprg kbhowto kbcode
KBSubcategory: BseSecurity BseMisc CodeSam

How to Select a Listview Item Programmatically in Windows 95

PSS ID Number: Q131284

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SUMMARY

Selecting a listview item in Windows 95 is not as easy as selecting a list box item was in Windows version 3.1. To select a list box item in Windows version 3.1, an application sends an LB_SETCURSEL or LB_SETSEL to a single- or multiple-selection list box respectively. To select a listview item in Windows 95, an application sends an LVM_SETITEMSTATE message or calls the ListView_SetItemState() macro.

MORE INFORMATION

An application can force a selection of a listview item. You might want the application to do this when a user clicks a column other than the first column of a listview of multiple subitems or columns.

Currently, a listview item is selected only when the user clicks the first column of that item. However, you many want the application to select the item regardless of which column in the listview is clicked.

Windows 95 does not provide a separate message or function to set the current selection in a listview. Instead, it defines item states or LVIS_* values that determine the listview item's appearance and functionality. LVIS_FOCUSED and LVIS_SELECTED in particular are the states that determine a listview item's selection state.

To select a listview item programmatically, an application sets the listview item's state as follows:

```
   ListView_SetItemState (hWndListView,         // handle to listview
                          iWhichItem,          // index to listview item
                          LVIS_FOCUSED | LVIS_SELECTED, // item state
                          0x000F);                      // mask
```

Note that the last parameter passed to this macro is a mask specifying which bits are about to change. LVIS_FOCUSED and LVIS_SELECTED are defined in <commctrl.h> as 0x0001 and 0x0002 respectively, so you need to set the last four bits of the mask.

The same principle applies to selecting a treeview item programmatically. The only difference is that an application sends a TVM_SETITEM message or

calls the TreeView_SetItem() macro.

Because listviews allow multiple selection by default, you can program an application to select multiple items by simulating a CTRL keydown (or SHIFT keydown event) prior to setting the item state. For example, the following code simulates the pressing of the CTRL key:

```
BYTE  pbKeyState [256];

GetKeyboardState ((LPBYTE)&pbKeyState);
pbKeyState[VK_CONTROL] |= 0x80;
SetKeyboardState ((LPBYTE)&pbKeyState);
```

Note that if an application simulates a keypress, it must also be responsible for releasing it by resetting the appropriate bit. For example, the following code simulates the release of a CTRL key:

```
BYTE  pbKeyState [256];

GetKeyboardState ((LPBYTE)&pbKeyState);
pbKeyState[VK_CONTROL] = 0;
SetKeyboardState ((LPBYTE)&pbKeyState);
```

Similarly, retrieving the currently selected item in a listview control in Windows 95 is not as easy as sending an LB_GETCURSEL message to a listbox control was in Windows version 3.1.

For listviews, call the ListView_GetNextItem() function with the LVNI_SELECTED flag specified:

```
iCurSel = ListView_GetNextItem (ghwndLV, -1, LVNI_SELECTED);
```

For treeviews, retrieve the currently selected item by calling the TreeView_GetNextItem() function with the TVGN_CARET flag specified or by calling the TreeView_GetSelection() macro directly:

```
iCurSel = TreeView_GetNextItem (ghwndTV, NULL, TVGN_CARET);
   or
iCurSel = TreeView_GetSelection (ghwndTV);
```

Additional reference words: 4.00 1.30
KBCategory: kbui kbcode
KBSubcategory: UsrCtl

How to Send Raw Data to a Printer by Using the Win32 API

PSS ID Number: Q138594

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SUMMARY
=======

It is sometimes necessary to send printer-specific data directly to a printer, bypassing the driver. The Win32 API provides a do it that works on local and networked printers. This method can be used to replace the PASSTHROUGH escape and SpoolFile() methods used in previous versions of the Windows API.

MORE INFORMATION
================

You can use the following code to send raw data directly to a printer in Windows NT or Windows 95.

```
// RawDataToPrinter - sends binary data directly to a printer
//
// Params:
//   szPrinterName - NULL terminated string specifying printer name
//   lpData        - Pointer to raw data bytes
//   dwCount       - Length of lpData in bytes
//
// Returns: TRUE for success, FALSE for failure
//
BOOL RawDataToPrinter( LPSTR szPrinterName, LPBYTE lpData, DWORD dwCount )
{
  HANDLE     hPrinter;
  DOC_INFO_1 DocInfo;
  DWORD      dwJob;
  DWORD      dwBytesWritten;

  // Need a handle to the printer
  if( ! OpenPrinter( szPrinterName, &hPrinter, NULL ) )
    return FALSE;

  // Fill in the structure with info about this "document"
  DocInfo.pDocName = "My Document";
  DocInfo.pOutputFile = NULL;
  DocInfo.pDatatype = "RAW";
  // Inform the spooler the document is beginning
  if( (dwJob = StartDocPrinter( hPrinter, 1, (LPSTR)&DocInfo )) == 0 )
  {
    ClosePrinter( hPrinter );
    return FALSE;
  }
  // Start a page
```

```
  if( ! StartPagePrinter( hPrinter ) )
  {
    EndDocPrinter( hPrinter );
    ClosePrinter( hPrinter );
    return FALSE;
  }
  // Send the data to the printer
  if( ! WritePrinter( hPrinter, lpData, dwCount, &dwBytesWritten ) )
  {
    EndPagePrinter( hPrinter );
    EndDocPrinter( hPrinter );
    ClosePrinter( hPrinter );
    return FALSE;
  }
  // End the page
  if( ! EndPagePrinter( hPrinter ) )
  {
    EndDocPrinter( hPrinter );
    ClosePrinter( hPrinter );
    return FALSE;
  }
  // Inform the spooler that the document is ending
  if( ! EndDocPrinter( hPrinter ) )
  {
    ClosePrinter( hPrinter );
    return FALSE;
  }
  // Tidy up the printer handle
  ClosePrinter( hPrinter );
  // Check to see if correct number of bytes writen
  if( dwBytesWritten != dwCount )
    return FALSE;
  return TRUE;
}
```

Additional reference words: 4.00 RAW.DRV binary
KBCategory: kbprint kbcode kbhowto
KBSubcategory: GdiPrn

How to Set Foreground/Background Responsiveness in Code

PSS ID Number: Q125660

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SUMMARY

=======

In Windows NT version 3.5, you can set foreground/background responsiveness by using the System Control Panel in Program Manager. Chose Tasking, then select one of the following through the dialog that is displayed:

- Best Foreground Application Response Time.
- Foreground Application More Responsive than Background.
- Foreground and Background Applications Equally Responsive.

This article describes how to achieve the same thing by using code in a program. It also explains how to override this setting by using code in a program.

MORE INFORMATION

================

You can use the Registry APIs to set foreground/background responsiveness. The following registry key allows you to specify the priority to give to the application running in the foreground:

```
HKEY_LOCAL_MACHINE\SYSTEM
   CurrentControlSet\
   Control\
   PriorityControl\
   Win32PrioritySeparation
```

The following values are supported:

Value	Meaning
0	Foreground and background applications equally responsive
1	Foreground application more responsive than background
2	Best foreground application response time

These values correspond to the choices offered in the Tasking dialog described in the "Summary" section of this article.

To override the setting from your application, use SetPriorityClass() to change your application's priority class and SetThreadPriority() to set the priority for a given thread.

NOTE: The thread priority together with the priority class for the process determine the thread's base priority.

Additional reference words: 3.50
KBCategory: kbprg
KBSubcategory: BseProcThrd

How To Set MCI Wave Audio Recording Format

PSS ID Number: Q152180

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK), version 3.1

- Microsoft Win32 Software Development Kit for
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Pegasus (Beta) 1.0

SUMMARY

All parameters in the MCI_WAVE_SET_PARMS struct that apply to recording wave audio should be set at the same time. If some recording parameters are set at one time and the balance of the recording parameters are set at a later time, recording will proceed at a default setting of 8-bits per sample, mono, and 11-kHz sampling.

MORE INFORMATION

The members of the MCI_WAVE_SET_PARMS struct that pertain to recording are wFormatTag, wBitsPerSample, nChannels, nSamplesPerSec, nAvgBytesPerSec, and nBlockAlign. All of these should be set in a single call to mciSendCommand in order to achieve recording under the desired settings.

Because nBlockAlign is usually computed using information from wBitsPerSample and nChannels, it may not seem necessary to set any members other than wFormatTag, wBitsPerSample, and nChannels in conjunction with the setting of nBlockAlign. Leaving the other two pertinent members, nSamplesPerSec and nAvgBytesPerSec, to be set later, however, results in an error message at both the initial setting and the later setting of parameters. The error message states "The parameter is out of range for the specified command." Recording will then proceed at a default setting of 8-bits per sample, mono, and 11-kHz sampling.

Following is a sample of setting all the recording members of the MCI_WAVE_SET_PARMS struct properly at once to get the desired 16-bit stereo recording accomplished at 44-kHz sampling:

```
   //#include <mmsystem.h> at the beginning of the program.
   //Link with mmsystem.lib for 16-bit code, or
   //link with winmm.lib for 32-bit code.

   MCI_WAVE_SET_PARMS set_parms;
   MCI_OPEN_PARMS          open_parms;
   DWORD                           dwReturn;
   UINT                                wave_device_id;
   char                                 buffer[128];
```

```
// Open the wave audio device.
open_parms.lpstrDeviceType = "waveaudio";
open_parms.lpstrElementName = "";

if (dwReturn = mciSendCommand( 0, MCI_OPEN, MCI_OPEN_TYPE |
      MCI_OPEN_ELEMENT, (DWORD) (LPVOID) &open_parms))
{
    mciGetErrorString(dwReturn, buffer, sizeof (buffer));
    MessageBox( NULL, buffer, "MCI_OPEN",
           MB_ICONEXCLAMATION | MB_OK);
}
else
{
    MessageBox( NULL, "Open Succeeded", "MCI_OPEN",
           MB_ICONEXCLAMATION | MB_OK);
}

// Note the wave audio device ID
wave_device_id = open_parms.wDeviceID;

// Set PCM format of recording.
set_parms.wFormatTag = WAVE_FORMAT_PCM;
set_parms.wBitsPerSample = 16;
set_parms.nChannels = 2;
set_parms.nSamplesPerSec = 44100;
set_parms.nAvgBytesPerSec = ((set_parms.wBitsPerSample)/8) *
                            set_parms.nChannels *
                            set_parms.nSamplesPerSec;
set_parms.nBlockAlign = ((set_parms.wBitsPerSample)/8) *
                            set_parms.nChannels;

if (dwReturn = mciSendCommand( wave_device_id, MCI_SET, MCI_WAIT |
                               MCI_WAVE_SET_FORMATTAG |
                               MCI_WAVE_SET_BITSPERSAMPLE |
                               MCI_WAVE_SET_CHANNELS |
                               MCI_WAVE_SET_SAMPLESPERSEC |

                               MCI_WAVE_SET_AVGBYTESPERSEC |
                               MCI_WAVE_SET_BLOCKALIGN,
                               (DWORD)(LPVOID)&set_parms))
{
    mciGetErrorString(dwReturn, buffer, sizeof(buffer));
    MessageBox( NULL, buffer, "MCI_SET",
           MB_ICONEXCLAMATION | MB_OK);
}
else
{
    MessageBox( NULL, "MCI_WAVE_SET Succeeded", "MCI_SET",
           MB_ICONEXCLAMATION | MB_OK);
}

//Close the wave device.
if (dwReturn = mciSendCommand( wave_device_id, MCI_CLOSE,
           (DWORD)NULL, (DWORD)NULL))
{
```

```
        mciGetErrorString(dwReturn, buffer, sizeof (buffer));
        MessageBox( NULL, buffer, "MCI_CLOSE",
               MB_ICONEXCLAMATION | MB_OK);
    }
    else
    {
        MessageBox( NULL, "MCI_CLOSE Succeeded", "MCI_CLOSE",
               MB_ICONEXCLAMATION | MB_OK);
    }
```

Following is behavior that will result in a default recording of 8-bits per sample, mono, and 11-kHz sampling, because the settings of the recording parameters are done in two parts:

```
    // Set PCM format recording, Part 1.
    set_parms.wFormatTag = WAVE_FORMAT_PCM;
    set_parms.wBitsPerSample = 16;
    set_parms.nChannels = 2;
    set_parms.nBlockAlign = ((set_parms.wBitsPerSample)/8) *
                               set_parms.nChannels;

    if (dwReturn = mciSendCommand( wave_device_id, MCI_SET, MCI_WAIT |
                                   MCI_WAVE_SET_FORMATTAG |
                                   MCI_WAVE_SET_BITSPERSAMPLE |
                                   MCI_WAVE_SET_CHANNELS |
                                   MCI_WAVE_SET_BLOCKALIGN,
                                   (DWORD)(LPVOID)&set_parms))
    {
        mciGetErrorString(dwReturn, buffer, sizeof(buffer));
        MessageBox( NULL, buffer, "MCI_WAVE_SET_1", MB_OK);
    }
    else
     }
        MessageBox( NULL, "MCI_WAVE_SET-1 Succeeded", "MCI_SET",
                                   MB_OK);
     }

     // Set PCM format recording, Part 2.
     set_parms.nSamplesPerSec = 44100;
     set_parms.nAvgBytesPerSec = ((set_parms.wBitsPerSample)/8) *
                                 set_parms.nChannels *
                                 set_parms.nSamplesPerSec;

     if (dwReturn = mciSendCommand( wave_device_id, MCI_SET, MCI_WAIT |
                                    MCI_WAVE_SET_SAMPLESPERSEC |
                                    MCI_WAVE_SET_AVGBYTESPERSEC,
                                    (DWORD)(LPVOID)&set_parms))
     {
         mciGetErrorString(dwReturn, buffer, sizeof(buffer));
         MessageBox( NULL, buffer, "MCI_WAVE_SET_2", MB_OK);
     }
     else
     {
         MessageBox( NULL, "MCI_WAVE_SET_2 Succeeded", "MCI_SET",
                                    MB_OK);
```

```
    }
```

Additional reference words: 3.11 4.00 3.51
KBCategory: kbmm kbsound kbhowto
KBSubcategory: MMWave

How to Set the Current Normal Vector in an OpenGL Application

PSS ID Number: Q131130

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SUMMARY

In a Microsoft Win32 OpenGL application, it is common practice to construct objects with the glBegin function followed by several calls to glVertex. For example, to create a flat polygon in three-dimensional space, you could write this code:

```
glBegin(GL_POLYGON);
glVertex3f(.....);
glVertex3f(.....);
glVertex3f(.....);
.  .  .
.  .  .
glEnd();
```

Now, if you want to implement a light source or multiple light sources in your OpenGL application, it is important that you include a call to the glNormal function between the calls to glBegin and glEnd so that the normal vector can be used by OpenGL when calculating the color to use when filling the polygon.

You can perform a vector cross product on two vectors to obtain a third vector that is perpendicular to the plane containing the two vectors. Using a vector cross product, you can calculate the vector normal to the polygon and use that value in your call to glNormal.

MORE INFORMATION

Using cross product math on two vectors of a polygon, you can obtain a vector that is perpendicular to the polygon. Two sides of a polygon describe two vectors in the plane of the polygon. With those two vectors, you can calculate a vector that is perpendicular to the polygon. The length of the normal vector calculated will not be unit length, and the normal vector needs to be unit length. Therefore, you need to call glEnable(GL_NORMALIZE) when you initialize your OpenGL application, so that normal vectors specified with glNormal are scaled to unit length after transformation.

Code Sample

You can use the following function to calculate a normal vector for a

polygon. You need to give it three points of the polygon and the points should be given in clock-wise order when you are facing the front of the polygon:

```
//*************************************************************************
// Function: CalculateVectorNormal
//
// Purpose: Given three points of a 3D plane, this function calculates
//          the normal vector of that plane.
//
// Parameters:
//     fVert1[]   == array for 1st point (3 elements are x, y, and z).
//     fVert2[]   == array for 2nd point (3 elements are x, y, and z).
//     fVert3[]   == array for 3rd point (3 elements are x, y, and z).
//
// Returns:
//     fNormalX   == X vector for the normal vector
//     fNormalY   == Y vector for the normal vector
//     fNormalZ   == Z vector for the normal vector
//
// Comments:
//
// History:  Date       Author        Reason
//           3/22/95     GGB           Created
//*************************************************************************

GLvoid CalculateVectorNormal(GLfloat fVert1[], GLfloat fVert2[],
                             GLfloat fVert3[], GLfloat *fNormalX,
                             GLfloat *fNormalY, GLfloat *fNormalZ)
    {
    GLfloat Qx, Qy, Qz, Px, Py, Pz;

    Qx = fVert2[0]-fVert1[0];
    Qy = fVert2[1]-fVert1[1];
    Qz = fVert2[2]-fVert1[2];
    Px = fVert3[0]-fVert1[0];
    Py = fVert3[1]-fVert1[1];
    Pz = fVert3[2]-fVert1[2];

    *fNormalX = Py*Qz - Pz*Qy;
    *fNormalY = Pz*Qx - Px*Qz;
    *fNormalZ = Px*Qy - Py*Qx;
}
```

Code to Call and Use the CalculateVectorNormal Function

Here is an example of how you might call and use the function:

```
   glBegin(GL_POLYGON);
   glVertex3fv(fVert1);
   glVertex3fv(fVert2);
   glVertex3fv(fVert3);
   glVertex3fv(fVert4);
```

```
   // Calculate the vector normal coming out of the 3D polygon.
   CalculateVectorNormal(fVert1, fVert2, fVert3, &fNormalX,
                         &fNormalY, &fNormalZ);
   // Set the normal vector for the polygon
   glNormal3f(fNormalX, fNormalY, fNormalZ);
   glEnd();
```

Additional reference words: 3.50 graphics
KBCategory: kbgraphic kbcode
KBSubcategory: GdiOpenGL

How to Set Up and Run the RNR Sample Included in the Win32 SDK

PSS ID Number: Q131505

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51

SUMMARY

This article explains how to set up and operate the Service Registration and Resolution (RNR) sample over the TCP and SPX network protocols. The RNR sample comes with the Microsoft Win32 (SDK) versions 3.5 and 3.51. It illustrates the use of the service registration and resolution APIs.

MORE INFORMATION

How to Set Up the RNR Sample

Run rnrsetup /ADD on each client and on the server to call the SetService API with SERVICE_ADD_TYPE. This call stores the service name type, its associated GUID, and relevant addressing information for the specified name spaces in the registry path:

```
\HKEY_LOCAL_MACHINE
  \CurrentControlSet
    \Control
      \ServiceProvider
        \ServiceTypes
```

This information is then retrieved by GetTypeByName() and GetAddressByName() calls respectively to identify the server's address.

How to Operate the RNR Sample

1. Start rnrsrv on one machine. After setting listening ports on the available protocols, the RNR server executes the SetService() call with the SERVICE_REGISTER flag to register the network service with the specified name spaces. For example, it enables advertising the EchoExample service name on the SAP protocol.

2. On the other machine, start rnrclnt using the following syntax:

   ```
   rnrclnt /?

   Usage: rnrclnt [/name:SVCNAME] [/type:TYPENAME] [/size:N]
             [/count:N]  [/rcvbuf:N] [/sndbuf:N]
   ```

   - TYPENAME is initially passed to GetTypeByName() call to return the GUID value. The GUID value and SVCNAME is then passed to

GetAddressByName() to return the address of the server that the client can connect to. TYPENAME is defined as EchoExample for the RNR sample.

- SVCNAME specifies which EchoExample server to connect to. If SVCNAME is specified as the server name in the Internet domain, the TCP protocol will be used. If SVCNAME is specified as EchoServer (the RNR service name advertised on SAP), the SPX protocol will be used.

- The other parameters to the rnrclnt have appropriate default values and are self explanatory.

Additional reference words: 3.50 3.51
KBCategory: kbnetwork kbnetwork
KBSubcategory: NtwkWinsock

How to Shade Images to Look Like Windows 95 Active Icon

PSS ID Number: Q128786

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

This article shows by example how to display an image or an icon in a shaded state, as Windows 95 does for the active icon.

MORE INFORMATION
================

Step-by-Step Procedure

To obtain the shaded look for your image or icon, follow these six steps:

1. Create a compatible DC and bitmap.

2. Create a monochrome pattern brush with every other pixel on.

3. Fill the memory image with the pattern.

4. BitBlt the source image over the pattern using SRCAND so that only the 'on' destination pixels are transferred.

5. Color the destination with the pattern, using the highlight color for the 'off' pixels and using black for the 'on' pixels.

6. Copy the filtered original from the memory DC to the destination using SRCPAINT so that only the 'on' pixels are transferred.

This results in the destination having the original image with every other pixel colored with the highlight color.

Sample Code

The following function implements these six steps to shade a rectangular area on a device context:

```
// ShadeRect
// hDC    : the DC on which the area is to be shaded
// lpRect : the coordinates within which to shade
BOOL ShadeRect( HDC hDC, LPRECT lpRect )
{
```

```
COLORREF  crHighlightColor, crOldBkColor, crOldTextColor;
HBRUSH    hBrush, hOldBrush;
HBITMAP   hBitmap, hBrushBitmap, hOldMemBitmap;
int       OldBkMode, nWidth, nHeight;
HDC       hMemDC;
RECT      rcRect = { 0, 0, 0, 0};
// The bitmap bits are for a monochrome "every-other-pixel"
//     bitmap (for a pattern brush)
WORD      Bits[8] = { 0x0055, 0x00aa, 0x0055, 0x00aa,
                      0x0055, 0x00aa, 0x0055, 0x00aa };

// The Width and Height of the target area
nWidth = lpRect->right - lpRect->left + 1;
nHeight = lpRect->bottom - lpRect->top + 1;

// Need a pattern bitmap
hBrushBitmap = CreateBitmap( 8, 8, 1, 1, &Bits );
// Need to store the original image
hBitmap = CreateCompatibleBitmap( hDC, nWidth, nHeight );
// Need a memory DC to work in
hMemDC = CreateCompatibleDC( hDC );
// Create the pattern brush
hBrush = CreatePatternBrush( hBrushBitmap );

// Has anything failed so far? If so, abort!
if( (hBrushBitmap==NULL) || (hBitmap==NULL) ||
    (hMemDC==NULL) || (hBrush==NULL) )
{
   if( hBrushBitmap != NULL ) DeleteObject(hBrushBitmap);
   if( hBitmap != NULL ) DeleteObject( hBitmap );
   if( hMemDC != NULL ) DeleteDC( hMemDC );
   if( hBrush != NULL ) DeleteObject( hBrush );
   return FALSE;
}

// Select the bitmap into the memory DC
hOldMemBitmap = SelectObject( hMemDC, hBitmap );

// How wide/tall is the original?
rcRect.right = nWidth;
rcRect.bottom = nHeight;

// Lay down the pattern in the memory DC
FillRect( hMemDC, &rcRect, hBrush );

// Fill in the non-color pixels with the original image
BitBlt( hMemDC, 0, 0, nWidth, nHeight, hDC,
lpRect->left, lpRect->top, SRCAND );

// For the "Shutdown" look, use black or gray here instead
crHighlightColor = GetSysColor( COLOR_HIGHLIGHT );

// Set the color scheme
crOldTextColor = SetTextColor( hDC, crHighlightColor );
crOldBkColor = SetBkColor( hDC, RGB(0,0,0) );
```

```
    SetBkMode( hDC, OPAQUE );

    // Select the pattern brush
    hOldBrush = SelectObject( hDC, hBrush );
    // Fill in the color pixels, and set the others to black
    FillRect( hDC, lpRect, hBrush );
    // Fill in the black ones with the original image
    BitBlt( hDC, lpRect->left, lpRect->top, nWidth, nHeight,
            hMemDC, 0, 0, SRCPAINT );

    // Restore target DC settings
    SetBkMode( hDC, OldBkMode );
    SetBkColor( hDC, crOldBkColor );
    SetTextColor( hDC, crOldTextColor );

    // Clean up
    SelectObject( hMemDC, hOldMemBitmap );
    DeleteObject( hBitmap );
    DeleteDC( hMemDC );
    DeleteObject( hBrushBitmap );
    SelectObject( hDC, hOldBrush );
    DeleteObject( hBrush );

    return TRUE;
}
```

Additional reference words: 4.00 hatch darken shadow
KBCategory: kbgraphic kbcode
KBSubcategory: GdiMisc

How to Share Data Between Different Mappings of a DLL

PSS ID Number: Q125677

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Under certain circumstances, 32-bit DLLs might have to share data with other 32-bit DLLs loaded by a different application or with different mappings of the same DLL. Because 32-bit DLLs are mapped into the the calling process's address space, which is private, sharing data with other DLLs mapped into the address spaces of different applications involves creating shared data section(s) or using memory mapped files. This article discusses the former -- creating shared data sections by using the #pragma statement. Typically, system-wide hooks installed in a DLL need to share some common data among different mappings.

MORE INFORMATION

Each Win32-based application runs in its own private address space. If a 32-bit application installs a system-wide hook with the hook callback function in a DLL, this DLL is mapped into the address space of every application for which the hook event occured.

Every application that the DLL gets mapped into, gets its own set of variables (data). Often there will be a scenario where hook callback functions mapped into different application or process address spaces need to share some data variables -- such as HHOOK or a Window Handle -- among all mappings of the DLL.

Because each application's address space is private, DLLs with hook callback functions mapped into one application's address spaces cannot share data (variables) with other hook callback functions mapped into a different application's address space unless a shared data SECTION exists in the DLL.

Every 32-bit DLL (or EXE) is composed of a collection of sections. Each section name begins with a period. The section of interest in this article is the data section. These sections can have one of the following attributes: READ, WRITE, SHARED, and EXECUTE.

DLLs that need to share data among different mappings can use the #pragma pre-processor command in the DLL source file to create a shared data section that contains the data to be shared.

The following sample code shows by example how to define a named-data

section (.sdata) in a DLL.

Sample Code

```
#pragma data_seg(".sdata")
int iSharedVar = 0;
#pragma data_seg()
```

The first line directs the compiler to place all the data declared in this section into the .sdata data segment. Therefore, the iSharedVar variable is stored in the MYSEC segment. By default, data is not shared. Note that you must initialize all data in the named section. The data_seg pragma applies only to initialized data. The third line, #pragma data_seg(), resets allocation to the default data section.

If one application makes any changes to variables in the shared data section, all mappings of this DLL will reflect the same changes, so you need to be carefule when dealing with shared data in applications or DLLs.

You must also tell the linker that the variables in the section you defined are to be shared by modifying your .DEF file to include a SECTIONS section or by specifying /SECTION:.sdata, RWS in your link line. Here's an example SECTIONS section:

```
   SECTIONS
   .sdata   READ WRITE SHARED
```

In the case of a typical hook DLL, the HHOOK, HINSTDLL, and other variables can go into the shared data section.

Additional reference words: 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

How to Simulate Changing the Font in a Message Box

PSS ID Number: Q68586

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

To simulate changing the font in a message box, create a dialog box that uses the desired font. Specify the style and contents of the dialog box to reflect the style of the desired message box. The application can also draw a system icon in the dialog box.

MORE INFORMATION

The message box is a unique object in Windows. Its handle is not available to an application; therefore, it cannot be modified. An application can simulate a message box with a different font by creating a dialog box that looks like a message box.

To change the font in a dialog box, use the optional statement FONT in the dialog statement of the resource script (.RC) file. For example, resource file statements for a dialog box displaying an error in Courier point size 12 would be as follows:

```
FontError DIALOG  45, 17, 143, 46
CAPTION "Font Error"
FONT 12, "Courier"
STYLE WS_CAPTION | WS_SYSMENU | DS_MODALFRAME
BEGIN
    CTEXT "Please select the right font", -1, 0, 7, 143, 9
    DEFPUSHBUTTON "OK"  IDOK,   56, 25,  32, 14, WS_GROUP
END
```

To center the dialog box in the screen, use GetWindowRect() to retrieve the dimensions of the screen and MoveWindow() to place the dialog box appropriately. The following code demonstrates this procedure:

```
case WM_INITDIALOG:
    GetWindowRect(hDlg, &rc);
    x = GetSystemMetrics(SM_CXSCREEN);
    y = GetSystemMetrics(SM_CYSCREEN);
    MoveWindow (hDlg,
       (x - (rc.right - rc.left)) >> 1,  /* x position */
       (y - (rc.bottom - rc.top)) >> 1,  /* y position */
```

```
          rc.right - rc.left,              /* x size */
          rc.bottom - rc.top,              /* y size */
          TRUE);                           /* paint the window */
       return TRUE;
```

To display a system icon in the dialog box, call the DrawIcon() function during the processing of a WM_PAINT message. After drawing the desired icon, the dialog procedure passes control back to the dialog manager by returning FALSE. The code to paint the exclamation point icon (used in warning messages) is as follows:

```
   case WM_PAINT:
       hIcon = LoadIcon(NULL, IDI_EXCLAMATION);
       hDC = GetDC(hDlg);
       DrawIcon(hDC, 20, 40, hIcon);
       ReleaseDC(hDlg, hDC);
       return FALSE;
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

How to Spawn a Console App and Redirect Standard Handles

PSS ID Number: Q126628

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

This article discusses spawning a console application with CreateProcess() and redirecting its output. The standard handles are controlled with the STARTUPINFO fields hStdInput, hStdOutput, and hStdError.

In Windows NT version 3.1, if a windowed application spawned a console application, you could:

- Redirect none of its standard handles (don't use STARTF_USESTDHANDLES).

  -or-

- Redirect all of its standard handles (use STARTF_USESTDHANDLES).

For example, if you redirected hStdInput and hStdOutput, but left hStdError as 0 or INVALID_HANDLE_VALUE, the console application would fail if it tried to write to stderr. This is not a problem for a console application spawning another console application.

In Windows NT version 3.5 and later and in Windows 95, if you set any of these fields to INVALID_HANDLE_VALUE, Windows NT will assign the default value to that handle in the console application, rather than leaving it an invalid value. Therefore, if you set STARTF_USESTDHANDLES, but fail to set one of the handle fields, this will not cause a problem for the console application. You can now redirect standard input, but not standard output, and so forth.

Additional reference words: 3.50 4.00
KBCategory: kbprg
KBSubcategory: BseProcThrd

How to Specify a Full Path in the ExecProgram Macro

PSS ID Number: Q86477

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.51 and 4.0

SUMMARY

Windows help files (.HLP files) can be written such that selecting a designated topic executes a Windows-based application. This is done with the ExecProgram() macro. If the desired application does not reside in the same directory as the .HLP file, a full path to the .EXE must be specified. An invalid path produces an error.

SYMPTOMS

Attempting to execute an application with a full path causes Windows to display the following Help message box:

```
Unable to Run Specified File.
```

CAUSE

A common mistake is to incorrectly specify the subdirectory delimiters in the path description.

RESOLUTION

The Windows Help Compiler can recognize escape sequences expressed using the backslash (\) character. When using the ExecProgram() macro, four backslashes (\\\\) separate each directory in a full path description:

```
ExecProgram( "c:\\\\winapps\\\\excel\\\\excel", 0 )
```

With HC31.EXE version 3.10.505 and HCW.EXE version 4.0, you would use only two backslashes:

```
ExecProgram( "c:\\winapps\\excel\\excel", 0 )
```

Additional reference words: 3.00 3.10 4.00 95
KBCategory: kbtool
KBSubcategory: TlsHlp

How to Specify Filenames/Paths in Viewer/WinHelp Commands

PSS ID Number: Q120251

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

Both WinHelp and Viewer have commands such as JumpId() that take a filename as a parameter. Depending on the circumstances under which such commands are called, the number of backslashes (\) included in the filename as part of the path may have to be doubled for the command to work correctly. For example, if JumpId() is called from a HotSpot, you need to use two backslashes to separate the subdirectory names as in this example:

```
JumpId("C:\\MYAPP\\HLPFILE\\MYHLP.HLP","topicx")
```

But if JumpId() is used as the command associated with a menu item, you need to use four backslashes to separate the subdirectory names within the JumpId command, which is itself part of an InsertItem() command. For example:

```
InsertItem("MNU_FILE","my_id","Jump",
   "JumpId('C:\\\\MYAPP\\\\HLPFILE\\\\MY HLP.HLP','topicx')",0)
```

To avoid having to modify the filename parameter depending on how the function is called, Microsoft recommends that you use a single forward slash (/) to separate subdirectory names within the filename. For example:

```
JumpId("C:/MYAPP/HLPFILE/MYHLP.HLP","topicx")
```

A single forward slash will work regardless of how the command is called.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsHlp

How to Specify Shared and Nonshared Data in a DLL

PSS ID Number: Q89817

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

To have both shared and nonshared data in a dynamic-link library (DLL) which is built with a 32-bit Microsoft C compiler, you need to use the #pragma data_seg directive to set up a new named section. You then must specify the correct sharing attributes for this new named data section in your .DEF file.

The system will try to load the shared memory block created by #pragma data_seg at the same address in each process. However, if the block cannot be loaded into the same memory address, the system allocates a new block of memory for that process and the memory is not shared. No run time warnings are given when this happens. Using memory-mapped files backed by pagefile (named shared memory) is a safer option than using #pragma data_seg, because the APIs will return an error when the mapping fails.

MORE INFORMATION

Below is a sample of how to define a named data section in your DLL. The first line directs the compiler to include all the data declared in this section in the .MYSEC data segment. This means that the iSharedVar variable would be considered part of the .MYSEC data segment. By default, data will be nonshared.

Note that you must initialize all data in your named section. The data_seg pragma only applies to initialized data.

The third line, "#pragma data_seg()", directs the compiler to reset allocation to the default data section.

```
#pragma data_seg(".MYSEC")
int iSharedVar = 0;
#pragma data_seg()
```

Below is a sample of the .DEF file that supports the shared and nonshared segments. This definition will set the default section .MYSEC to be shared. The default data section is by default non-shared, so any data not in section .MYSEC will be non-shared.

```
LIBRARY MyDll
SECTIONS
```

```
     .MYSEC READ WRITE SHARED
   EXPORTS
    ...
```

NOTE: All section names must begin with a period character ('.') and must not be longer than 8 characters, including the period character.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseDll

How to Start a Control Panel Applet in Windows 95

PSS ID Number: Q135068

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

In Windows 95, the Control Panel application (Control.exe) is just a 16-bit application stub that calls into the shell to start the Control Panel. To debug a control panel applet, the programmer needs to invoke the Control Panel as the calling application, but Control.exe is a 16-bit application, so the debugger cannot load it. To invoke a control panel applet in Windows 95, a program can call into the shell directly.

MORE INFORMATION

There's a utility shipped with Windows 95 called RunDLL32 that allows you to invoke a function exported from a DLL. The Windows 95 shell doesn't call Control.exe when you start the control panel, instead it invokes a Control_RunDLL function in Shell32.DLL. With this information, you can use with the following command line to invoke a control panel applet:

```
rundll32.exe shell32.dll,Control_RunDLL mycontrol.cpl
```

This starts the first control panel applet in Mycontrol.cpl. If you have multiple control panel applets in Mycontrol.cpl, you need to add to the line as shown here:

```
rundll32.exe shell32.dll,Control_RunDLL mycontrol.cpl,@1
```

Here @1 specifies the second (zero-based) applet in the .cpl file. If you don't specify this parameter, @0 is used as the default.

There's one more parameter you can add. It serves as the command line parameters passed to the control panel applet in the CPL_STARTWPARM notification. For example, a lot of the system's control panel applets take the page number (one based, not zero based) as the command line parameter. For example, if you want to start the Add/Remove Programs applet at the Windows Setup page, so you can instruct the user to add extra system components, you can use this code:

```
rundll32.exe shell32.dll,Control_RunDLL appwiz.cpl,@0,2
```

NOTE: If you put a space after the comma in the commands above, you will get the error:

```
Error in shell32.dll
```

Missing entry.

Additional reference words: Windows 95 4.00 debug CPL
KBCategory: kbui
KBSubcategory: UsrExt

How to Start an Application at Boot Time Under Windows 95

PSS ID Number: Q125714

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY
=======

Windows NT supports a Win32-based application type known as a Service. A Service may be started at boot time, automatically, by a user with the Service Control Manager facility or by Win32-based applications that use the service-related Win32 APIs. The Service Control subsystem and the associated Win32 APIs are not supported in Windows 95. In place of services, Windows 95 has two registry keys that will allow users to run applications before a user logs in when the system first starts up.

MORE INFORMATION
================

Microsoft recognizes the value that Services and the Service Control manager have, therefore, we have implemented a smaller version of the Service Control Manager in Windows 95 so that applications can run before a user logs in. At boot time, the system checks two new registry keys: "RunServices" and "RunServicesOnce".

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion
   RunServices  [key]
      bubba95=service.exe /params [string value]

   ...

   RunServicesOnce  [key]

   ...
```

The value names are arbitrary. The value data is the command line passed to CreateProcess(). Values under the key RunServicesOnce are deleted after the application is launched. Because these applications are started before the user logs onto the system, the user has not been validated and the applications cannot assume that they have particular networking permissions enabled. Windows 95, unlike Windows NT, only has one security context for the entire system. Therefore, don't assume that any application that starts has access to a particular network resource because a particular user has access to this network resource.

Applications started by the RunServices and RunServicesOnce keys will be closed when the user selects "Close all programs and log on as a different user" from the Shutdown dialog on the Start Menu. A Win32-based application can prevent itself or any other Win32-based application from being closed

when the user logs off by calling RegisterServiceProcess(). Win32-based applications registered in this manner close only when the system is shut down. However, the application must consider that different user can be logged on at different times during its execution. The application can distinguish between a user logging off and the system shutting down by examining the lParam of WM_QUERYENDSESSION and WM_ENDSESSION. If the system is being shut down, lParam is NULL. If the user is logging off, lParam is set to EWX_REALLYLOGOFF.

To access RegisterServiceProcess, retrieve a function pointer using GetProcAddress() on KERNEL32.DLL.

RegisterServiceProcess

```
DWORD RegisterServiceProcess(
    DWORD dwProcessId,   // process identifier
    DWORD dwServiceType  // type of service
   );
```

Parameters:

- dwProcessId: Specifies the identifier of the process to register as a service process or NULL to register the current process. An application receives the process ID when it uses CreateProcess() to start the application.

- dwServiceType: One of the following values:

Define	Value	Meaning
RSP_SIMPLE_SERVICE	0x00000001	Registers the process as a simple service process.
RSP_UNREGISTER_SERVICE	0x00000000	Unregisters the process as a service process.

Return Value:

The return value is 1 if successful or 0 if an error occurs.

Additional reference words: 3.95 4.00
KBCategory: kbprg
KBSubcategory: BseService

How to Stop a Journal Playback

PSS ID Number: Q98486

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

To stop a "journal playback" when a specified key is pressed, the filter function must determine whether the key was pressed and then call the UnhookWindowsHookEx function to remove the WH_JOURNALPLAYBACK hook.

MORE INFORMATION
================

To determine the state of a specified key, the filter function must call the GetAsyncKeyState function when the nCode parameter equals HC_SKIP. The HC_SKIP hook code notifies the WH_JOURNALPLAYBACK filter function that Windows is done processing the current event.

The GetAsyncKeyState function determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to the GetAsyncKeyState function. If the most significant bit of the return value is set, the key is down; if the least significant bit is set, the key was pressed after a preceding GetAsyncKeyState call.

If the filter function calls the GetAsyncKeyState function after the specified key was pressed and released, then the most significant bit will not be set to reflect a key-down. Thus, a test to check whether the specified key is down fails. Therefore, the least significant bit of the return value must be checked to determine whether the specified key was pressed after a preceding call to GetAsyncKeyState function. Using this technique of checking the least significant bit requires a call to the GetAsyncKeyState function before setting the WH_JOURNALPLAYBACK hook. For example:

```
// When setting the journal playback hook.
   .
   .
   .
   // Reset the lease significant bit.
   GetAsyncKeyState( VK_CANCEL );

   // Set a system-wide journal playback hook.
```

```
   g_hJP = SetWindowsHookEx( WH_JOURNALPLAYBACK,
            FilterFunc,
            g_hInstDLLModule,
            NULL );
   .
   .
   .

// Inside the filter function
   .
   .
   .
   if ( nCode == HC_SKIP )
       if ( GetAsyncKeyState( VK_CANCEL) )
           UnhookWindowsHookEx( g_hJP );
   .
   .
   .
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrHks

How to Subclass a Window in Windows 95

PSS ID Number: Q125680

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY
=======

While subclassing windows within the same application in Windows 95 is unchanged from Windows version 3.1, subclassing windows belonging to other applications is somewhat more complicated in Windows 95. This article explains the process.

MORE INFORMATION
================

For a 16-bit application, subclassing methods are the same as they were in Windows version 3.1. However, Windows 95 performs some behind-the-scenes magic to make it possible for a 16-bit window to subclass a 32-bit window.

Usually, a subclass consists of saving one window procedure and substituting another in its place. However, this could present a problem when a 16-bit application tries to call a 32-bit window procedure. Windows 95 works around this potential problem by providing 32-bit windows with a 16-bit window procedure. All 32-bit windows will have the same selector for their wndProcs that references code in KRNL386.EXE where the 16-bit wndProcs for all 32-bit windows are stored. Eventually, each of these 16-bit wndProcs will jump to the real 32-bit window procedure.

Subclassing windows belonging to another process, either 16-bit or 32-bit, from a 32-bit process or application works as it does in Windows NT. The difficulty is that each 32-bit process has its own private address space. Hence, a window procedure's address in one process is not valid in another. To get a window procedure from one process into another, you need to inject the subclass procedure code into the other process's address space. There are a number of ways to do this.

Three Ways to Inject Code Into Another Process's Address Space
---

Method 1: Windows 95 and Windows NT

You can use the registry, hooks, or remote threads and the WriteProcessMemory() API to inject code into another process's address space.

Method 2: Windows NT Only

If you use the registry, the code that needs to be injected should reside in a DLL. By either running REGEDIT.EXE or using the registry APIs, add the \HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\AppInit_DLLs key to the registry if it does not exist. Set its value to a string containing the DLL's pathname. This key may contain more than one DLL pathname separated by single spaces. This has the effect, once the machine is restarted, of loading the library with DLL_PROCESS_ATTACH into every process at its creation time. While this method is very easy, it also has several disadvantages. For example, the computer must be restarted before it takes effect, and the DLL will last the lifetime of the process.

Method 3: Windows 95 and Windows NT

You can also use hooks to inject code into another process's address space. When a window hooks another thread belonging to a different process, the system maps the DLL containing the hook procedure into the address space of the hooked thread. Windows will map the entire DLL, not just the hook procedure. So to subclass a window in another process, install a WH_GETMESSAGE hook or another such hook on the thread that owns the window to be subclassed. In the DLL that contains the hook procedure, include the subclass window procedure. In the hook procedure, call SetWindowLong() to enact the subclass. It is important to leave the hook in place until the subclass is no longer needed, so the DLL remains in the target window's address space. When the subclass is removed, the hook would be unhooked, thus unmapping the DLL.

A third way to inject a DLL into another address space involves the use of remote threads and the WriteProcessMemory() API. It is more flexible and significantly more complicated than the previously mentioned methods, and is described in the following reference.

REFERENCES
==========

"Load Your 32-bit DLL into Another Process's Address Space Using INJLIB" by Jeffrey Richter, MSJ May 1994.

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrWndw

How to Support Language Independent Strings in Event Logging

PSS ID Number: Q125661

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

Insertion strings in the event log entries are language-independent. Instead of using string literals as the insertion string, use "%%n" as the insertion string.

MORE INFORMATION

When the event viewer sees "%%n", it looks up the ParameterMessageFile value in the registry, under the source of the event, as in this example:

```
HKEY_LOCAL_MACHINE\SYSTEM\
   CurrentControlSet\
   Services\
   EventLog\
   Security\
   ...

-or-

HKEY_LOCAL_MACHINE\SYSTEM\
   CurrentControlSet\
   Services\
   EventLog\
   System\
   Service Control Manager
```

It then calls the LoadLibrary() function of the ParameterMessageFile. Then it calls FormatMessage() using "n" as the ID.

For example, suppose an event log entry has the source "Service Control Manager" and the description is "Failed to start the service due to the following error: %%245."

In the registry, you find:

```
HKEY_LOCAL_MACHINE\SYSTEM\
   CurrentControlSet\
   Services\
   EventLog\
   System\
   Service Control Manager
```

```
        EventMessageFile...
        ParameterMessageFile REG_SZ  kernel32.dll
        TypesSupported...
        ...
```

Therefore, you need to follow these steps:

1. Use LoadLibrary() with KERNEL32.DLL.

2. Call FormatMessage() using the module handle obtained in step 1 and a
   string ID of 245.

3. Replace %%245 in the description with the string obtained in step 2.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

How to Test Autorun.inf Files

PSS ID Number: Q136214

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SUMMARY
=======

AutoPlay is enabled by the new 32-bit, protected-mode driver architecture in Windows 95. Because the operating system can now detect the insertion of media in a CD-ROM drive, it has the opportunity to do some intelligent processing whenever this occurs.

By default, Windows 95 only checks for an Autorun.inf file when a CD-ROM disc is inserted into the CD-ROM drive. However, you may want to test for syntax and logic errors before burning a CD-ROM disc. This article explains how.

MORE INFORMATION
================

In Windows 95, whenever a CD-ROM disc is inserted, the shell immediately checks to see if the CD-ROM disc has a PC filesystem. If it does, Windows 95 looks for a file named Autorun.inf. If the file exists, Windows 95 follows the instructions contained in the file, which usually involves running a setup application of some sort.

However, by changing a setting in the Windows 95 registry, you can have the Shell use AutoPlay on any media, including shared network drives and floppy disks.

The registry key that needs to be modified is:

```
HKEY_CURRENT_USER\
   Software\
      Microsoft\
         Windows\
            CurrentVersion\
               Policies\
                  Explorer\
                     "NoDriveTypeAutoRun"
```

This key, which is of type REG_BINARY, consists of four bytes. The first byte is a bitmask defining which drive types should be AutoRun. The other three bytes should be set to zero (0).

The bits in the bitmask correspond to these constants:

Type	Bit
DRIVE_UNKNOWN	0

```
DRIVE_NO_ROOT_DIR   1
DRIVE_REMOVABLE     2
DRIVE_FIXED         3
DRIVE_REMOTE        4
DRIVE_CDROM         5
DRIVE_RAMDISK       6
```

Setting a bit in the bitmask prevents you from using AutoPlay with the corresponding drive type. By default, the value in the registry is 0x95. Bits 0, 2, 4 and 7 are therefore set, which means that drive types DRIVE_UNKNOWN, DRIVE_REMOVEABLE, and DRIVE_REMOTE don't use AutoPlay information. (Bit 7 is set to cover future device types.) Altering this registry value thus allows you to test Autorun.inf files from a floppy disk (DRIVE_REMOVEABLE), network drive (DRIVE_REMOTE), and so on. For example, to be able to test AutoPlay from a floppy disk, set the value of the first byte to 0x91 to enable AutoPlay for floppy disks.

NOTE: Most floppy disk drive controllers do not currently recognize when a floppy disk has been inserted. To test AutoPlay on a floppy disk, first alter the registry setting so that bit DRIVE_REMOVEABLE is not set, start the Windows Explorer, insert the floppy disk, and press the F5 key to refresh the display.

To test the Autorun.inf file on a given disk, using the right mouse button, click the icon for the drive in the Windows Explorer. The effects of the Autorun.inf file should be visible in the context menu.

Additional reference words: 4.00
KBCategory: kbmm
KBSubcategory: MMCDROM

How to Toggle the NUM LOCK, CAPS LOCK, and SCROLL LOCK Keys

PSS ID Number: Q127190

The information in this article applies to:

 - Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The documentation for SetKeyboardState() correctly says that you cannot use this API to toggle the NUM LOCK, CAPS LOCK, and SCROLL LOCK keys.

You can use keybd_event() to toggle the NUM LOCK, CAPS LOCK, and SCROLL LOCK keys under Windows NT. The same technique works for toggling CAPS LOCK and SCROLL LOCK under Windows 95, but it does not work for NUM LOCK.

MORE INFORMATION
================

The following sample program turns the NUM LOCK light on if it is off. The SetNumLock function defined here simulates pressing the NUM LOCK key, using keybd_event() with a virtual key of VK_NUMLOCK. It takes a boolean value that indicates whether the light should be turned off (FALSE) or on (TRUE).

The same technique can be used for the CAPS LOCK key (VK_CAPITAL) and the SCROLL LOCK key (VK_SCROLL).

Sample Code

```
/* Compile options needed:
*/

#include <windows.h>

void SetNumLock( BOOL bState )
{
   BYTE keyState[256];

   GetKeyboardState((LPBYTE)&keyState);
   if( (bState && !(keyState[VK_NUMLOCK] & 1)) ||
       (!bState && (keyState[VK_NUMLOCK] & 1)) )
   {
   // Simulate a key press
      keybd_event( VK_NUMLOCK,
                   0x45,
                   KEYEVENTF_EXTENDEDKEY | 0,
                   0 );
```

```
   // Simulate a key release
      keybd_event( VK_NUMLOCK,
                   0x45,
                   KEYEVENTF_EXTENDEDKEY | KEYEVENTF_KEYUP,
                   0);
   }
}

void main()
{
   SetNumLock( TRUE );
}
```

Additional reference words: 3.50 4.00 95
KBCategory: kbui kbcode
KBSubcategory: UsrInp

How to Troubleshoot Win32s Installation Problems

PSS ID Number: Q106715

The information in this article applies to:

- Microsoft Win32s versions 1.1, 1.15, 1.2, 1.25a

SUMMARY
=======

The installation guide for Win32s that is included in the Win32 SDK recommends running Freecell to verify that the installation was successful. This article discusses some of the errors that may occur when trying to run Freecell on an unsuccessful installation. The article also contains a list of corrective actions to help you reinstall Win32s.

MORE INFORMATION
================

Possible Symptoms

One of the following may occur when running Freecell if the installation is not successful:

- File Error: Cannot find Olecli.dll

  -or-

- Win32s - Error:
  Improper installation. Win32s requires Win32s.exe and Win32s16.dll to run. Reinstall Win32s.

  -or-

- Win32s - Error:
  Improper installation. Windows requires W32s.386 in order to run. Reinstall Win32s.

  -or-

- Error: Cannot find file Freecell.exe (or one of its components)...

  -or-

- The display is corrupted as soon as you run FreeCell.

  -or-

- Win32s Error:
  One of the System Components is out of date. Please re-install the application.

Possible Step-by-Step Solution

Use the following steps when Win32s does not install correctly:

1. If you are having video problems, check to see if you have an S3 video card. Certain S3 drivers do not work with Win32s. Either use the generic drivers shipped with Windows or contact your video card manufacturer for an updated driver. For additional information on the S3 driver and Win32s, please see the following article in the Microsoft Knowledge base:

   ARTICLE-ID: Q117153
   TITLE : PRB: Display Problems with Win32s and the S3 Driver

2. Make sure the following line is in your System.ini file:

   device=*vmcpd

3. If you have a printer driver by LaserMaster, delete it or change it into a comment; it interferes with installing Win32s. Then reboot the computer so the changes will take effect. After you successfully reinstall Win32s, reinstall the driver or remove the comment characters.

   The driver interferes with installing Win32s because the LaserMaster drivers create a WINSPOOL device. The extension is ignored when the file name portion of a path matches a device name. As a result, when Setup tries to write to Winspool.drv, it fails. It fails because it attempts to write to WINSPOOL. In fact, any Win32-based application that tries to link to Winspool.drv also fails; however, most Win32-based applications that print under Win32s do not use the WINSPOOL application programming interfaces (APIs) because they are not supported in Win32s. As a result, you can usually just disable this driver while installing Win32s, and then reenable it afterwards. To disable the driver, you need to turn the following three lines in the [386ENH] of the System.ini file into comments.

      device = lmharold.386
      device = lmmi
      device = lmcap

4. Delete the \Win32s directory, the \Win32app directory, W32sys.dll, w32s16.dll, and Win32s.exe from your hard drive before installing. Although it is possible to install Win32s on top of an old installation of Win32s, it is better to remove the old files before installing the new ones.

   Edit the Win32s.ini file on your hard drive. Change SETUP = 1 to read SETUP = 0. Reboot your computer and reinstall Win32s.

5. Make sure that paging is enabled. From the Control Panel, select the 386 Enhanced icon, choose Virtual Memory, and choose Change. Verify that the drive type is not set to none. The type can be set to either temporary or permanent.

6. If you are using SHARE (not Vshare.386, which Windows for Workgroups uses), make sure that SHARE is enabled. Edit the Autoexec.bat file, and add the following line if it is not already there:

   C:\Dos\Share.exe

What to Do If the Previous Steps Don't Solve the Problem

If you still receive errors when running Freecell, compare the binaries on your hard drive with those on the CD-ROM. Use the MS-DOS program Filecomp.exe, Comp.exe, or Fc.exe and do a binary compare. For example, if you have the Win32 SDK CD-ROM, type the following:

   fc /b <system>\win32s\w32s.386 <cd>:\mstools\win32s\nodebug\w32s.386

If you have the 32-bit Visual C++ CD-ROM, type the following:

   fc /b <system>\win32s\w32s.386 <cd>:\msvc32s\win32s\retail\w32s.386

The results of the compare will be "FC: no differences encountered" if the binaries were correctly installed. If the binaries are not the same, you might have a bad copy of the files or a bad compact disc.

As a last resort, you can try reinstalling Win32s that shipped in the Visual C++ 32-bit edition. There seems to be a slight difference between the Setup.exe program on the Win32 SDK CD-ROM and the Setup.exe on the Visual C++ CD-ROM. If you try installing from the 32-bit Visual C++ CD-ROM, don't remove the \Win32app or \Freecell directory or any of the Freecell files. The Visual C++ CD-ROM does not contain Freecell.

NOTE: Using Freecell Help will generate errors, including "Routine Not Found" or "Help Topic Does Not Exist." The generation of these errors has nothing to do with whether or not Win32s is installed correctly.

Freecell.hlp was meant to be used with Winhlp32.exe. Freecell.hlp uses the advanced features of WINHLP32 for full-text searching. Winhelp.exe, which runs under Windows version 3.1, does not support this. As a result, each time Freecell.hlp tries to bind the Find button to the full-text searching APIs, it fails, and Windows Help displays the message box. You can still read the information in the help file, and you can use the Search button to do keyword searches.

If you are getting the Win32s Error "One of the System Components is out of date. Please re-install the application," you might be installing from the Win32s Upgrade (Pw1118.exe). The Setup.exe program that comes with Pw1118.exe is meant to be run ONLY under the retail version of Win32s. If you have switched to the debug version of Win32s in the past by using Switch.bat file in the Win32s\Bin directory and then tried to run the Setup.exe program from Pw1118.exe, Win32s will not be installed properly on your computer. This is because some of the Win32s files will not be replaced correctly.

Similary, the Switch.bat program in the Win32s\Bin directory cannot be used to upgrade your previous installation of Win32s.

There are two solutions for this problem:

- Switch to the retail version of Win32s first by using the Switch.bat file in the Win32s\Bin directory. Then run the Setup.exe program from Pw1118.exe.

  -or-

- Remove your previous installation of Win32s as described in this article, and then run the Setup.exe program from Pw1118.exe. For more information on how to delete the previous version of Win32s from your computer, please see the following article in the Microsoft Knowledge Base:

  ARTICLE-ID: Q120486
  TITLE : How to Remove Win32s

Additional reference words: 1.10 1.20
KBCategory: kbsetup kbtshoot kbwebcontent
KBSubcategory: W32s

How to Update the List of Files in the Common Dialogs

PSS ID Number: Q109696

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

SUMMARY

=======

Sometimes it is necessary to update the list of files, without terminating the dialog box, when using the File Open or Save As common dialog box. This can be done by simulating a double-click on the list box of directories. Although the message can be posted from any application, a hook procedure should be used to post the message to the dialog box window.

MORE INFORMATION

================

The common dialog box functions that update the list of files and directories are internal to the common dialog boxes and are not accessible by applications using the common dialog box routines. The functions are invoked and the list boxes are updated only when the user double-clicks a list box.

Sample Code

The following code uses the Cancel button of the common dialog boxes to update the list boxes:

```
BOOL CALLBACK __export FileOpenHook (HWND hDlg, UINT message,
                        WPARAM wParam, LPARAM lParam)
{
   switch (message)
   {
      case WM_COMMAND:
         switch(wParam)
         {
         // This simulates a double-click on the list of directories,
         // effectively forcing the common dialogs to re-read the current
         // directory of files and to refresh the list of files.
            case IDCANCEL :
               PostMessage( hDlg, WM_COMMAND, lst2,
                           MAKELPARAM(GetDlgItem(hDlg, lst2), LBN_DBLCLK);
               return TRUE;
         }
      break;
   }
```

```
   return FALSE;
}
```

If the application targets Win32, the notification message to the list box is sent differently; here is the PostMessage for Win32 applications:

```
PostMessage (hDlg, WM_COMMAND, MAKEWPARAM (lst2,LBN_DBLCLK),
          (LPARAM)GetDlgItem (hDlg, lst2));
```

Applications using the IDs of the common dialog box's controls must include the DLGS.H file.

The templates for the common dialog boxes are in the \SAMPLES\COMMDLG directory or in the \INCLUDE directory of the Windows SDK installation.

Additional reference words: 3.10 3.50 refresh redraw
KBCategory: kbui
KBSubcategory: UsrCmnDlg

How to Use 32-bit StampResource() Function in Windows NT 3.51

PSS ID Number: Q133699

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SUMMARY

=======

The 32-bit setup toolkit function, StampResource(), is used to write string table resource data into binary (executable) files. This function is useful for customizing installed files with installation-specific resource data, such as a user's name, company, and so on. This article discusses issues specific to using this function in the 32-bit Setup Toolkit for Windows NT version 3.51.

MORE INFORMATION

================

StampResource() is prototyped in the 32-bit Setup Toolkit as follows:

```
VOID StampResource(LPSTR szSect, LPSTR szKey, LPSTR szDst,
   int wResType, int wResId, LPSTR szData, int cbData);
```

Notice that the szData argument is prototyped as LPSTR. All string resource data in Windows NT must be stored in UNICODE(tm) form. Therefore, because StampResource() is writing directly into a binary file, you must convert the string data for StampResource() into wide-character format. That means you must make your szData string parameter type LPWSTR. The following example shows how this can be done:

```
INT size;
TCHAR buf[80], szName[40],szCompany[20],szProductID[20];
LPWSTR wUnicodeString;

lstrcpy(szName,"Bob Smith");
lstrcpy(szCompany,"Microsoft");
lstrcpy(szProductID,"123-45-6789");
wsprintf(buf,"%c%s%c%s%c%s",lstrlen(szName),szName,lstrlen(szCompany),
   szCompany,lstrlen(szProductID),szProductID);
size=MultiByteToWideChar(CP_ACP,MB_PRECOMPOSED,buf,-1,NULL,0);
wUnicodeString=(LPWSTR)GlobalAlloc(GMEM_FIXED,sizeof(WCHAR)*size);
MultiByteToWideChar(CP_ACP,MB_PRECOMPOSED,buf,-1,wUnicodeString,size);
StampResource("Extra Files","Config","d:\\setup32\\stamp\\disks\\disk1",
   6,0x451,(LPSTR)wUnicodeString,sizeof(WCHAR)*size);
GlobalFree((HGLOBAL)wUnicodeString);
```

In this example, the code performs these steps:

1. Convert the ASCII string "buf" to wide-character format by using the Win32 MultiByteToWideChar() function. This first call to this function retrieves the size, in wide characters, needed to store the resultant

   wide-character string.

2. Allocate memory for the resultant wide-character string and perform the
   actual conversion by calling MultiByteToWideChar() again.

3. Cast the wide-character string "wUnicodeString" to type LPSTR during the
   call to StampResource() to avoid a type mismatch warning message.

NOTE: this information does not apply to Windows 95. The 32-bit
StampResource() function does not currently work in Windows 95
because it makes use of a Win32 API function that is not supported by
Windows 95.

REFERENCES
==========

For more information on the other parameters of StampResource(), or how
StampResource() works in general, please see the following article in the
Microsoft Knowledge Base:

   ARTICLE-ID: Q92525
   TITLE   : Using the Setup Toolkit Function StampResource

Additional reference words: 3.51
KBCategory: kbtool
KBSubcategory: tlsmss

How to Use a DIB Stored as a Windows Resource

PSS ID Number: Q67883

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Device-independent bitmaps (DIBs) are a very useful tool for displaying graphic information in a variety of device environments. With the appropriate device drivers, Windows can display a DIB with varying results on any video display or on a graphics printer.

This article discusses the differences between two methods that can be used to access a DIB from a resource.

MORE INFORMATION

Bitmaps retrieved from resources are very similar to those stored in .BMP files on disk. The header information is the same for each type of bitmap. However, depending upon the method used to retrieve the bitmap from the resource, the bitmap may be a device-independent bitmap (DIB) or a device-dependent bitmap (DDB).

When the LoadBitmap() function is used to obtain a bitmap from a resource, the bitmap is converted to a DDB. Typically, the DDB will be selected into a memory device context (DC) and blt'ed to the screen for display.

NOTE: If a 256-color bitmap with a palette is loaded from a resource, some colors will be lost. To display a bitmap with a palette correctly, the palette must be selected into the destination DC before the image is transferred to the DC. LoadBitmap() cannot return the palette associated with the bitmap; therefore, this information is lost. Instead, the colors in the bitmap are mapped to colors available in the default system palette, and a bitmap with the system default color depth is returned.

For example, if LoadBitmap() loads a 256-color image into an application running on a VGA display, the 256 colors used in the bitmap will be mapped to the 16 available colors, and a 4 bits-per-pixel bitmap will be returned. When the display is a 256-color 8514 unit, the same action will map the 256 bitmap colors into the 20 reserved system colors, and an 8 bits-per-pixel bitmap will be returned.

If, instead of calling LoadBitmap(), the application calls FindResource() (with RT_BITMAP type), LoadResource(), and LockResource(), a pointer to a packed DIB will be the result. A packed DIB is a BITMAPINFO structure followed by an array of bytes containing the bitmap bits.

NOTE: If the resource was originally stored as a DDB, the bitmap returned will be in the DDB format. In other words, no conversion is done.

The BITMAPINFO structure is a BITMAPINFOHEADER structure and an array of RGBQUADs that define the colors used in the DIB. The pointer to the packed DIB may be used in the same manner as a bitmap read from disk.

NOTE: The BITMAPFILEHEADER structure is NOT present in the packed DIB; however, it is present in a DIB read from disk.

REFERENCES
==========

For sample code demonstrating how to use FindResource() with RT_BITMAP, LoadResource(), and LockResource(), please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q124947
   TITLE   : Retrieving Palette Information from a Bitmap Resource

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiBmp

How to Use a File Created by an Apple Macintosh in Windows NT

PSS ID Number: Q147438

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API)
  included with Microsoft Windows NT, versions 3.5 and 3.51

NOTE: Some products mentioned in this article are manufactured by vendors independent of Microsoft; we make no warranty, implied or otherwise, regarding these products' performance or reliability.

SUMMARY

Windows NT Server has a set of services named "Services For Macintosh (SFM)" that allow Apple Macintosh computers to transparently gain access to shared files and printers that reside on Windows NT servers. Windows NT Server also allows PC clients to gain access to these same shared files and printers. When Macintosh clients connect to a shared directory on a Windows NT server, SFM presents the files as though they were native to Macintosh computers. Likewise, when PC clients connect to the shared directory, the Windows NT server presents the files as though they were native to PC computers.

The file system used by the Macintosh Operating System is very different from file systems supported by Windows NT. As a result, file naming conventions and the structure of files differ considerably between these two systems. This article shows how SFM and Windows NT work together to hide some of these details, and explains how Win32 applications can properly handle files native to Macintosh clients.

MORE INFORMATION

File Server for Macintosh (FSM), which is part of SFM, allows Windows and Macintosh clients to create and open files on the same share by presenting the files to the clients with the naming conventions used by their operating systems. That is, regardless of which client actually created a file, it appears as a Macintosh file to Macintosh clients and a Windows file to Windows clients.

File Names

Some file names cannot be represented exactly when viewed by applications from both Macintosh and Windows systems because the Macintosh and Windows operating systems use different code pages to represent character strings and different file naming conventions. File Server for Macintosh works with NTFS to preserve the file semantics used by both systems by translating illegal characters in file names so that both Macintosh and Windows clients may access the same files.

Regardless of which type of client creates a file, NTFS stores its file name on disk in UNICODE format. When a Macintosh client creates a file on a share, FSM translates characters in the file name that are invalid on NTFS into equivalent characters in the user-defined range of the UNICODE character set so that the file name is legal on NTFS.

When a Macintosh client uses a file that it or another Macintosh client created on an NTFS share, it will see the original name. File Server for Macintosh maps the characters on disk from the UNICODE user-defined range back into their original Macintosh-specific characters.

When a Macintosh client gains access to a file created by a Windows client and that file doesn't follow the Macintosh file-naming standards, FSM maps the invalid characters into the equivalent characters in the Macintosh code page. The Macintosh client will then see the file with substituted characters in its name.

UNICODE-enabled Win32 applications will see file names with characters in the user-defined space and can open them without a problem. Windows-based applications that use the ANSI character set will see a second mapping from the user-defined UNICODE characters to the ANSI default character -- the question mark (?). Because the question mark is also a wild-card character on Windows platforms, Windows applications that use the ANSI character set will not be able to open these files.

NTFS will automatically generate short file name equivalents for long file names created by both Macintosh and Win32 clients so that 16-bit Windows and MS-DOS clients can gain access to them. The rules for automatic short name generation are the same as Windows NT uses any time a long file name is created on disk.

File Structures

Windows platforms represent files differently from the Macintosh Operating System. File Server for Macintosh handles these differences so that Macintosh clients see files as though they were native to Macintosh computers, and Windows clients see them as being native to Windows platforms. This section explains how files are represented on both operating systems and how FSM handles them.

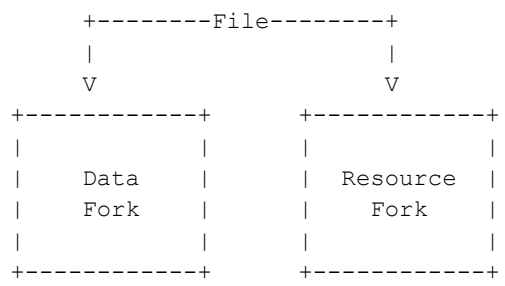
Windows platforms do not define the structure or interpretation of any file's content; applications can impose any structure on the content. Except for a very limited case, all files consist of a single stream (linear sequence) of bytes. These files are called "flat" because they don't have any higher-level organization. Win32 applications running on Windows NT can create files with multiple streams on NTFS volumes only. However, as a general rule, they should not create files with multiple streams because they will be "flattened" when copied to volumes with the FAT file system.

Because Win32 platforms can use different file systems, the maximum file size depends on which file system is used. Files on the Protected-Mode FAT file system may have a maximum size of 2 gigabytes (GB). On NTFS, the largest file can be 17 billion GB long (2 to the 64th power bytes). Only

Win32 applications running on Windows NT can access the full range of files larger than 2 GB.

The Macintosh Operating System uses files consisting of two linear sequences of bytes known as forks. The data fork is used to store the file's data and corresponds to a file created by a Win32 platform. The resource fork is used to store information about the file, such as its icon or the fonts used in it. Every file has both forks, but either or both may be empty.

As far as the Macintosh Operating System is concerned, the content of the data fork is unstructured and is subject to interpretation by applications. The resource fork has a definite structure imposed by the Macintosh Operating System and is used for file management purposes. The following diagram shows how a file created by a Macintosh is structured:



When a file created by a Macintosh is stored on a Windows NT server, it is stored in a single file that contains a stream for each fork. When Windows-based applications gain access to a file created by a Macintosh client, File Server for Macintosh presents only the data fork because the data fork is the equivalent of the representation of files on Windows platforms. Windows-based applications may not access the resource fork.

The Macintosh Operating System allows files and volumes to be as large as 2 GB.

Design Recommendations for Win32 Applications

Some Win32-based applications running on a Windows NT server need to manage files that were created by Macintosh clients. Although File Server for Macintosh presents files as each type of client expects to use them, applications that manage files must do so correctly to preserve the file names and structure for Macintosh clients. This section presents the four most important design guidelines that you should follow to handle files properly.

1. Handle file names with UNICODE strings. This can be done either by making the application completely UNICODE-enabled or by explicitly calling the UNICODE versions of the file APIs. (UNICODE APIs end with a "W" as in CopyFileW.)

   Win32 applications that use file names of files created by Macintosh clients must use UNICODE strings to prevent problems in translating characters that are unique to the Macintosh. Because file names are

stored on disk in UNICODE, when a UNICODE application uses them, the file name the application sees is identical to that stored on the disk. The application can find, copy, or move the file without forcing its name to be translated.

Non-UNICODE Windows-based applications that retrieve and then store files by name (example operations include find, copy, and move) will force file names to be translated from UNICODE to ANSI and then back to UNICODE. During the first part of the translation, characters in the UNICODE user-defined space are mapped to their nearest equivalents in the ANSI code page. In the second part, the ANSI characters are written back out to disk. This process results in a file name that may not exactly match the original because the ANSI characters written are the results of the mapping, not the original characters.

2. Don't copy files created by Macintosh clients to volumes that have the FAT file system. The FAT file system does not support multiple streams, so the file will be flattened; that is, its resource fork will be lost.

3. Don't copy or rename files created by Macintosh clients from Windows 95, Win32s, or Windows 3.x clients. When these clients connect to the Windows NT server through a redirected drive letter or UNC name, they gain access to shared files through the clients' disk I/O systems. The problem with these clients is that their disk I/O systems do not support multiple file streams, so the files they save will be flattened, as described in guideline 2.

   To allow Windows 95, Win32s, and Windows 3.x clients to gain access to files created by Macintosh clients, the server should provide an RPC server or service to handle requests. The RPC server or service should then use the UNICODE file APIs as described in guideline 1.

4. Ensure that Win32-based applications running under Windows NT do not create files larger than 2 GB because Macintosh clients won't be able to use them.

File Name Reference

A code page is a mapping of the numerical bytes (character codes) used to represent strings and their representation on the display (glyphs). Because Win32 and Macintosh systems use different code pages, some extended ASCII characters on one system are not representable on screen in exactly the same way on the other. For example, on the Macintosh, the glyph for character code 0xBD is the Greek Omega character while in the Windows ANSI code page, it is character symbol for "1/2."

The Macintosh Operating System uses the following file naming conventions:

- File and folder (directory) names may be up to 31 characters long; full pathnames are not limited to 260 characters (as they are on Win32 platforms).

- The colon (:) is used as a path separator.

- File and folder names may use any characters except the colon, which is the path separator. Characters such as ?, *, \, and / are perfectly legal in Macintosh file names.

- Although the Macintosh Operating System preserves the case of all file names, it does not distinguish between file names by case. (That is, file and folder names are not case-sensitive.)

Windows Platforms use two different file systems -- the Protected-Mode File Allocation Table (FAT) file system and the New Technology File System (NTFS). These file systems have similar naming conventions, and differ mainly in which characters are illegal. The naming conventions listed below apply to both FAT and NTFS unless specifically noted.

- File and directory names may be up to 255 characters long on FAT file systems and 256 characters on NTFS. Full pathnames may be up to 260 characters long.

- The backslash (\) is the path separator.

- File and directory names on the Protected-Mode FAT file system may consist of letters, digits, spaces, and these characters:

  $%'-_@~`!{}()#&+,;=[].

  Note that periods are allowed in file and directory names, as long as they are accompanied by other characters. For example, .text is perfectly legal.

  On NTFS, file and directory names may consist of any character except the following characters:

  "/*?<>|:

- Although file and directory names are not case-sensitive, their case is preserved.

REFERENCES
==========

"Inside Macintosh: Files," copyright 1992 Apple Computer, Inc.

"Services For Macintosh," Microsoft Windows NT Server product documentation, copyright 1985-1994 Microsoft Corporation.

"File Systems," Microsoft Win32 SDK overview, copyright 1985-1986 Microsoft Corporation.

Additional reference words: 3.51 3.50 SFM Mac fork
KBCategory: kbprg kb3rdparty kbhowto
KBSubcategory: BseFileio

How to Use a Program to Calculate Print Margins

PSS ID Number: Q122037

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

SUMMARY

The Windows Software Development Kit (SDK) does not provide a function to calculate printer margins directly. An application can calculate this information using a combination of printer escapes and calls to the GetDeviceCaps() function in Windows or by using GetDeviceCaps() in Windows NT. This article discusses those functions and provides code fragments as illustrations.

MORE INFORMATION

An application can determine printer margins as follows:

1. Calculate the left and top margins

   a. Determine the upper left corner of the printable area by using the GETPRINTINGOFFSET printer escape in Windows or by calling GetDeviceCaps() with the PHYSICALOFFSETX and PHYSICALOFFSETY indices in Windows NT. For example:

```
// Init our pt struct in case escape not supported
pt.x = 0; pt.y = 0;

// In Windows NT, the following 2 calls replace GETPRINTINGOFFSET:
// pt.x = GetDeviceCaps(hPrnDC, PHYSICALOFFSETX);
// pt.y = GetDeviceCaps(hPrnDC, PHYSICALOFFSETY);

// In Windows, use GETPRINTINGOFFSET to fill the POINT struct
// Drivers are not required to support the GETPRINTINGOFFSET escape,
// so call the QUERYESCSUPPORT printer escape to make sure
// it is supported.
Escape (hPrnDC, GETPRINTINGOFFSET, NULL, NULL, (LPPOINT) &pt);
```

   b. Determine the number of pixels required to yield the desired margin (x and y offsets) by calling GetDeviceCaps() using the LOGPIXELSX and LOGPIXELSY flags.

```
// Figure out how much you need to offset output. Note the
// use of the "max" macro. It is possible that you are asking for
// margins that are not possible on this printer. For example, the HP
// LaserJet has a 0.25" unprintable area so we cannot get margins of
// 0.1".
```

```
      xOffset = max (0, GetDeviceCaps (hPrnDC, LOGPIXELSX) *
                        nInchesWeWant - pt.x);

      yOffset = max (0, GetDeviceCaps (hPrnDC, LOGPIXELSY) *
                        nInchesWeWant - pt.y);

   // When doing all the output, you can either offset it by the above
   // values or call SetViewportOrg() to set the point (0,0) at
   // the margin offset you calculated.

   SetViewportOrg (hPrnDC, xOffset, yOffset);
   ... all other output here ...
```

2. calculate the bottom and right margins

   a. Obtain the total size of the physical page (including printable and unprintable areas) by using the GETPHYSPAGESIZE printer escape in Windows or by calling GetDeviceCaps() with the PHYSICALWIDTH and PHYSICALHEIGHT indices in Windows NT.

   b. Determine the number of pixels required to yield the desired right and bottom margins by calling GetDeviceCaps using the LOGPIXELSX and LOGPIXELSY flags.

   c. Calculate the size of the printable area with GetDeviceCaps() using the HORZRES and VERTRES flags.

   The following code fragment illustrates steps 2a through 2c:

```
   // In Windows NT, the following 2 calls replace GETPHYSPAGESIZE
   // pt.x = GetDeviceCaps(hPrnDC, PHYSICALWIDTH);
   // pt.y = GetDeviceCaps(hPrnDC, PHYSICALHEIGHT);

   // In Windows, use GETPHYSPAGESIZE to fill the POINT struct
   // Drivers are not required to support the GETPHYSPAGESIZE escape,
   // so call the QUERYESCSUPPORT printer escape to make sure
   // it is supported.
   Escape (hPrnDC, GETPHYSPAGESIZE, NULL, NULL, (LPPOINT) &pt);

   xOffsetOfRightMargin = xOffset +
                          GetDeviceCaps (hPrnDC, HORZRES) -
                          pt.x -
                          GetDeviceCaps (hPrnDC, LOGPIXELSX) *
                          wInchesWeWant;

   yOffsetOfBottomMargin = yOffset +
                           GetDeviceCaps (hPrnDC, VERTRES) -
                           pt.y -
                           GetDeviceCaps (hPrnDC, LOGPIXELSY) *
                           wInchesWeWant;
```

NOTE: Now, you can clip all output to the rectangle bounded by xOffset, yOffset, xOffsetOfRightMargin, and yOffsetOfBottomMargin.

For further information about margins, query in the Microsoft Knowledge Base by using these words:

   GETPHYSPAGESIZE and GETPRINTINGOFFSET and GetDeviceCaps

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbprint kbprg kbcode
KBSubcategory: GdiPrn

How to Use CTL3D Under the Windows 95 Operating System

PSS ID Number: Q130693

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY
=======

When an application that uses CTL3D is run under Windows 95, CTL3D disables itself if any dialog box has the DS_3DLOOK style. By default, all applications based on Windows version 4.0 get the DS_3DLOOK style for all dialog boxes. This article explains how this affects the way dialog boxes and controls are displayed under the Windows 95 operating system.

MORE INFORMATION
================

When CTL3D is disabled, Windows 95 draws dialog boxes and controls using its own 3D drawing properties. Windows 95 does not draw the static rectangles and frames in 3D as CTL3D does. For more information about how these frames and rectangles are drawn under Windows 95, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q125684
TITLE   : How to Use SS_GRAYRECT SS_BLACKRECT SS_WHITERECT in Windows 95

There are two new static control styles (SS_SUNKEN and SS_ETCHEDFRAME) in Windows 95 that simulate two of the static panels used in CTL3D. SS_SUNKEN creates a sunken panel, and SS_ETCHEDFRAME creates a panel with a dipped edge. There is no static style for creating a raised panel, but you can use the DrawEdge API to draw a raised panel.

There are also two new static control styles that you can use to create 3D lines. SS_ETCHEDHORZ creates a dipped horizontal line, and SS_ETCHEDVERT creates a dipped vertical line.

An application should check the platform version at run time by using the GetVersion or GetVersionEx function, and then implement appropriate 3D effects. If the major version is less than 4, the application can use the CTL3D functions, messages, and controls. If the major version is 4 or greater, the application should not implement CTL3D; it should create the proper Windows 95 style controls (or use DrawEdge to draw its 3D panels) to achieve the desired effects.

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrCtl

How to Use DWL_MSGRESULT in Property Sheets & Wizard Controls

PSS ID Number: Q130762

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SUMMARY

Each page in a property sheet or wizard control is an application-defined modeless dialog box that manages the control windows used to view and edit the properties of an item. Applications provide the dialog box template used to create each page as well as the dialog box procedure.

A property sheet or wizard control sends notification messages to the dialog box procedure for a page when the page is gaining or losing the focus and when the user chooses the OK, Cancel, or other buttons. The notifications are sent in the form of WM_NOTIFY messages. The dialog box procedure(s) for the correspoding page(s) should use the SetWindowLong() function to set the DWL_MSGRESULT value of the page dialog box to specify the return value from the dialog box procedure to prevent or accecpt the change. After doing so, the dialog box procedure must return TRUE in response to processing the WM_NOTIFY message. If it does not return TRUE, the return value set in the DWL_MSGRESULT index using the SetWindowLong() function is ignored by the property sheet or wizard control.

MORE INFORMATION

Dialog box procedures return a BOOL value (TRUE or FALSE). This return value indicates to the caller of the dialog box function that the dialog box function either handled the message that it received or did not handle it. When the dialog box function returns FALSE, it is indicating that it did not handle the message it received. When the dialog box handles the message and generates a return value, it typically sets the DWL_MSGRESULT index of the dialog box with the return value.

The dialog box function of the property sheet or wizard page handles messages (WM_NOTIFY) sent by the property sheet or wizard control. The property sheet or wizard control determines whether the page that received the message processed the message or not by checking the retrun value from the call to SendMessage(). If the return value is FALSE, the control goes ahead and does what needs to be done by default. But if the return value is TRUE, the control checks for the return value by looking at the value stored in the DWL_MSGRESULT index of that page.

For example, the dialog box function of a property page might trap the PSN_SETACTIVE notification to prevent it from being activated under certain circumstances. In this case, the page dialog box function uses the SetWindowLong() function to set the DWL_MSGRESULT value to -1. If the

dialog box does not return TRUE after setting the DWL_MSGRESULT, the property sheet control that sent the message completely ignores the return value because it assumes there is no return value.

Additional reference words: 4.00 common controls user Windows 95
KBCategory: kbui
KBSubcategory: UsrCtl

How to use ExitExecRestart to Install System Files

PSS ID Number: Q114606

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, version 3.5

SUMMARY

Some installation procedures require the installation of files (such as CTL3D.DLL, COMMDLG.DLL, and fonts) that may be in use by Windows at the time the setup program is run. Windows is likely to have these files open, so they cannot be installed without causing sharing violations. The Setup Toolkit provides features to exit Windows, install these files, and then restart Windows when complete.

MORE INFORMATION

The Setup Toolkit accomplishes the installation of system files as follows:

1. Before the Setup Toolkit copies a system file, it checks to see if the file is currently open. If it is, it copies the file to the destination directory, but under a different file name. It then adds this file to the "restart list".

2. When CopyFilesInCopyList() is complete, the Setup Toolkit checks the "restart list" and generates a .BAT file (named _MSSETUP.BAT) in the "restart directory". This .BAT file contains commands which delete the system files which were open (in step #1) and rename the new versions to their correct names.

3. Windows is exited, the .BAT file executed, then Windows restarted.

4. The .BAT file is then deleted.

   NOTE: The "restart directory" is not deleted. Hence, you should use your application's installation directory as your restart directory.

Hence, to install system files, perform the following steps:

1. Mark the system files as "system" in the DSKLAYT program. This is accomplished by highlighting all the system files (clicking with the CTRL key down) and placing a check in the "System File" check box under "File Attributes".

2. Before calling CopyFilesInCopyList() specify the name of your "restart directory". Assuming the target directory for your application is stored in DEST$ (as in the samples), use the following line:

```
   SetRestartDir DEST$
```

The specified directory does not need to exist. It will be created if necessary.

3. After your installation is complete, execute the following code before exiting your setup script. Normally this code will be placed at the end of the Install subroutine.

```
   if RestartListEmpty ()=0 then
       '' The following two lines must go on one line.
       MessageBox hwndFrame (), "Windows will now be exited and
       restarted.", "Sample Setup Script", MB_OK+MB_ICONINFORMATION
   eer:
       i%=ExitExecRestart ()
       '' The following three lines must go on one line.
       MessageBox hwndFrame (), "Windows cannot be restared because
       MS-DOS-based applications are active. Close all MS-DOS-based
       applications, and then click OK.", "Sample Setup Script",
       MB_OK+MB_ICONSTOP
       goto eer
   end if
```

NOTE: In order to use the MessageBox() function you must add the following lines at the beginning of your setup script:

```
   const MB_ICONINFORMATION = 64
   '' The following two lines must go on one line.
   declare sub MessageBox lib "user.exe" (hwnd%, message$,
   title$, options%)
```

4. Add the file _MSSETUP.EXE to your source directory and lay it out on Disk #1 in DSKLAYT.

5. Add a reference to _MSSETUP.EXE to the [files] section of your .LST file. For example,

    if you marked _MSSETUP.EXE to be compressed,

```
    [files]
        _mssetup.ex_ = _mssetup.exe
```

    if you did not mark it as compressed,

```
    [files]
        _mssetup.exe = _mssetup.exe
```

NOTES:

1. If ExitExecRestart () is successful, your script will be exited. That is, ExitExecRestart () will not return. If it does return, an error has occurred.

2. This functionality is not available under Windows 3.0. If the user runs the above setup script on Windows 3.0, they will receive the message

that MS-DOS-based applications are running and they will not be able to complete the setup. If this is a concern, check the version of Windows before executing the above code.

3. If _MSSETUP.EXE is not in your .LST file or not laid out in DSKLAYT, you will receive an "assertion failure" message when calling ExitExecRestart ().

Additional reference words: 3.10 3.50 setup toolkit mssetup
KBCategory: kbtool kbsetup kbprg kbcode
KBSubcategory: TlsMss

How to Use Hangeul (Korean) Windows Input Method Editor (IME)

PSS ID Number: Q130053

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Win32s version 1.2

SUMMARY

Hangeul (Korean) Windows supports the KSC5601-1987 code set, which consists of several thousands Hangeul characters. The Hangeul Windows IME (Input Method Editor) allows the user to enter Hangeul Jamos (24 basic components of Hangeul characters), compose the final characters, and send them to applications.

MORE INFORMATION

When running Hangeul Windows, you will see a small window with three buttons in the lower-left corner:

- The left button toggles between the Roman "A" character and the Korean character that is pronounced "GA," which Windows uses to symbolize Korean characters.

- The center button is a graphic of a half box or full box, which selects SBCS or DBCS storage (or display) of Roman (but not Hangeul) characters. Hangeul characters always take up a double-byte space, unlike some Japanese characters (katakana).

- The right button is the "Hangeul to Hanja" converter.

You can type in English by having the Roman toggle selected. To try typing Hangeul, the main thing to remember is that there is no apparent logical connection between the US 101 keyboard and what the you end up typing. To type like a pro, you need Korean keycaps, a cheat sheet, or a lot of memory in your brain.

For example, to type the word "Hangeul" type these characters:

   GKS   RMF

(Please ignore the spaces between the two characters.) Notice how each group of three keyboard characters assemble a single Hangeul character. Hangeul is often made up of three components (called "Jamos"), but characters can actually be composed of from two to several Jamos.

Here is how to type "Seoul, Korea." Seoul is pronounced locally "sa-ul," so try typing these characters:

TJ DNF ZH FL DK

(Please ignore the spaces between the characters.) The word "Korea" is not a Korean name; it is the English equivalent, just as Japan is really Nippon. The word "Korea" in Korean is "Han-guk," so "Seoul, Hanguk" would be:

TJD NF GKS RNR

The reason to try "Hanguk" instead of "Korea" is that Hanguk can also be spelled with Chinese characters. Put the mouse pointer (cursor) on the left edge of "Han" and click the "Hangeul to Hanja" button. A list box appears. Select choice #1 by typing 1 or by selecting it with the mouse. Do the same for the next character, again selecting choice #1. Then it will display in Hanja.

Most Korean text is in Hangeul not Hanja, notice the 3.1-H readme is all Hangeul. Hanja is still used in Korea - sometimes.

Additional reference words: 1.20 3.10 3.50 Hangul kbinf
KBCategory: kbother
KBSubcategory: wintldev

How to Use IDFEDIT to Create MIDI Instrument Definition Files

PSS ID Number: Q134994

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0

SUMMARY

Idfedit.exe is a software tool and sample included in the Win32 SDK. You can use it to create MIDI Instrument Definition Files (IDF) for the MIDI Mapper in Windows 95. The Windows 95 MIDI Mapper also supports the Windows version 3.1 Configuration file format (CFG) for backwards compatibility, but Microsoft recommends that any new MIDI Mapper configurations be created using the IDF file format. The Windows NT version 3.51 MIDI Mapper does not yet support the IDF file format, so it still requires the old CFG file format.

MORE INFORMATION

You can use Idfedit to create new IDF files or to edit existing ones. To change values in Definition Data fields, double click the data field, and enter the new data. It is important to note that some of the fields have restrictions on the kind of data they accept. The following table summarizes the required data types for each field:

Field	Data type
IDF Version	hexadecimal
IDF Creator	hexadecimal
Manufacturer	ASCII text string
Product Name	ASCII text string
Supports General MIDI	boolean values (expressed as TRUE, FALSE, YES, NO, 1 or 0)
Supports SysEx Messages	boolean values (expressed as TRUE, FALSE, YES, NO, 1 or 0)

The remaining fields accept only decimal values.

Other than the source code provided for Idfedit, there is currently no additional source of information about the IDF file format. The sample code is located in the \Mstools\Samples\Mm\Win95\Idfedit directory.

Additional reference words: 4.00 Windows 95
KBCategory: kbmm kbprg
KBSubcategory: MMMidi

How to Use LANA Numbers in a 32-bit Environment

PSS ID Number: Q138037

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.1, 3.5, 3.51, 4.0

SUMMARY
=======

NetBIOS uses the concept of a LANA (LAN adapter number) that allows you to write transport-independent NetBIOS applications. This article describes what a LANA is and recommends an approach to writing NetBIOS applications.

MORE INFORMATION
================

A LANA is a field of the NetBIOS NCB structure. In IBM's NetBIOS 3.0 specification, a LANA was used to specify a particular network adapter, as NetBIOS then supported up to two network adapters in one PC computer. Specifying a LANA of zero directed a request to the first adapter, and specifying a LANA of one directed a request to the second adapter.

Originally, IBM sent NetBIOS packets over the NETBEUI protocol, also known as the NetBIOS Frames protocol. This was the only transport NetBIOS could use to send data across the network. In other words, each network adapter had only one protocol to send and receive NetBIOS packets.

Because most computers have only one network adapter, many MS-DOS-based applications send all their requests to a LANA value of zero (also called simply 'LANA zero'). If a second network adapter is installed, some programs allow the user to configure the application to use LANA one instead. As a result, LANA zero became a default setting, though it was never intended to be a default.

Today's network technology allows NetBIOS to use transports other than NETBEUI. Microsoft has extended the meaning of LANA to indicate a specific transport on a specific adapter. For example, if you have two network adapters, and have IPX/SPX and NETBEUI transports installed, you have four LANAs. The LANAs may or may not be sequential, and there is no systematic way to identify which transport maps to which LANA.

In addition to extending the meaning of a LANA, Microsoft also added an NCB command (NCBENUM) that returns an array of available LANA numbers. As an example, the LANA_ENUM structure filled by NCBENUM might hold an array with values 0, 3, 5, and 6. Zero might map to IPX/SPX on the first adapter, three might map to NETBEUI on a second adapter, and so on.

In Windows NT and Windows 95, network adapters consist of physical adapters (like a 3Com Etherlink II) and software adapters (like the Dial Up Adapter). In addition, a user may have TCP/IP, NETBEUI, IPX/SPX, and other transports installed, all of which have NetBIOS support.

For Windows NT, LANAs are configurable through the control panel. Choose the Network applet, choose the NetBIOS Interface component, then choose Configure. A dialog appears that allows you to edit the LANAs.

For Windows 95, you may only set LANA zero, the default protocol, and if no protocol is set as default, there won't be a LANA zero. You can set the default protocol in the control panel. Choose the Network applet, choose the protocol you want as default, choose Properties, the Advanced tab, and finally check 'Set this protocol to be the default protocol'.

LANAs may seem like a constraint that your application must work around. However, making your application ignorant of how users want to configure their machines is a powerful idea, and one that makes life easier for your customers.

The best way to write a NetBIOS application is to support all LANAs, and establish connections over any LANA. A good approach is outlined in the following steps:

1. Enumerate the LANAs by submitting NCBENUM.

2. Reset each LANA by submitting one NCBRESET per LANA.

3. Add your local NetBIOS name to each LANA. The name may be the same on each LANA.

4. Connect using any LANA:

    - For servers, submit an NCBLISTEN on each LANA. If necessary, cancel any outstanding listen after the first listen is satisfied.

    - For clients, submit an NCBFINDNAME on each LANA. The first successful find name will give you a LANA on which you can submit an NCBCALL.

It is a good idea to submit NCBADDNAME, NCBLISTEN, and NCBFINDNAME asynchronously. Asynchronous requests will be processed almost in parallel on each transport.

This architecture is quite beneficial. Once your application is written to establish connections in this manner, it will support any transport that NetBIOS can use. As a result, your customers will not have to configure anything within your application, and your application will not be affected by dynamic LANAs such as dial-up adapters or plug-and-play hardware.

REFERENCES
==========

IBM NetBIOS specification version 3.0 and Win32 SDK Documentation.

Additional reference words: LANA 3.10 3.50 3.51 4.00
KBCategory: kbnetwork kbprg
KBSubcategory: NtwkNetBios

How to Use LVIF_DI_SETITEM on an LVN_GETDISPINFO Notification

PSS ID Number: Q131285

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SUMMARY

Windows 95 provides two flags, LVIF_DI_SETITEM and TVIF_DI_SETITEM, for the listview and treeview controls respectively. When set, these flags instruct Windows to start storing information for that particular item previously set as a callback item.

MORE INFORMATION

Windows 95 introduces the concept of callback items for the new listview and treeview common controls. A callback item is a listview or treeview item for which the application, not the control, stores the text, icon, or any appropriate information about the item. If the application already maintains this information anyway, setting up callback items could decrease the memory requirements of the control. Callback items are just as useful for items that display constantly changing information. Setting these items up as callback items allows the application to display the most current values appropriate for that item.

Take for example a SPY application that displays information in a hierarchical form (or a treeview) about the window being browsed or spied on. One of the things it displays is the window rectangle, or the dimensions of the window.

Because the user could resize this window at any time, this particular item is a good candidate for a callback item because it displays constantly changing information.

The application defines a callback item by specifying LPSTR_TEXTCALLBACK for the pszText member of the TV_ITEM structure. Whenever the item needs to be displayed, Windows requests the callback information by sending the treeview's parent a TVN_GETDISPINFO notification in the form of a WM_NOTIFY message. The parent window then fills the pszText member of the TV_ITEM structure as the following sample code demonstrates:

```
   LRESULT MsgNotify(HWND   hwnd,
                     UINT   uMessage,
                     WPARAM wparam,
                     LPARAM lparam)
   {
```

```
      TV_DISPINFO *ptvdi = (TV_DISPINFO *)lparam;

      switch (ptvdi->hdr.code)
      {
         case TVN_GETDISPINFO:

           if (ptvdi.mask & TVIF_TEXT)
           {
              RECT  rect;
              char  szBuf [30];

              GetWindowRect (hWndToBrowse, &rect);

              // where FormatRectText formats the rect information
              // in a nice <WindowRect: (x,y):cx,cy> format
              // and stores it in szBuf.
              FormatRectText (&rect, szBuf, sizeof (szBuf));

              lstrcpy (ptvdi.pszText, szBuf);
           }
           :

          default: break;

       }
   return 0;
   }
```

At a certain point, the application may determine during run time, that the window dimensions will no longer change. At this point, there may be no reason for this particular treeview item to remain as a callback item. This time, you need to process the TVN_GETDISPINFO message in a slightly different manner, specifying the TVIF_DI_SETITEM flag as demonstrated in the following code:

```
   case TVN_GETDISPINFO:

      if (ptvdi.mask & TVIF_TEXT)
      {
         RECT  rect;
         char  szBuf [30];

         GetWindowRect (hWndToBrowse, &rect);

        // where FormatRectText formats the rect information
        // in a nice <WindowRect: (x,y):cx,cy> format
        // and stores it in szBuf.
         FormatRectText (&rect, szBuf, sizeof (szBuf));

         lstrcpy (ptvdi.pszText, szBuf);
         plvdi->item.mask = plvdi->item.mask | TVIF_DI_SETITEM;
      }
```

By ORing the mask with TVIF_DI_SETITEM, you instruct Windows to start storing text information for the particular treeview item. At that point,

the application stops receiving a TVN_GETDISPINFO notification whenever the item needs to be redrawn. This works almost as well as calling TreeView_SetItem() on the item and replacing pszText's value with LPSTR_TEXTCALLBACK to the appropriate string.

The same holds true for listview controls when the mask is ORed with the LVIF_DI_SETITEM flag. However, note that setting the LVIF_DI_SETITEM flag for listviews works only for the first column of text (iSubItem ==0).

If an application specifies LPSTR_TEXTCALLBACK therefore for a column other than 0 in report view, LVIF_DI_SETITEM does not store the text information for that listview item column.

For more information on other members of the LV_ITEM and TV_ITEM structures that can be set up for callback, refer to the documentation on LV_DISPINFO and TV_DISPINFO structures.

Additional reference words: 4.00 1.30 I_IMAGECALLBACK win95 I_CHILDRENCALLBACK
KBCategory: kbui kbcode
KBSubcategory: UsrCtl

How to Use MIDI Stream Callbacks

PSS ID Number: Q142107

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SUMMARY
=======

Applications running under Windows 95 that use MIDI streams to send events to a MIDI device can use MIDI stream callbacks to be notified of the playback progress at specified times during playback of the stream. This article discusses the following topics regarding MIDI stream callbacks:

- Uses of MIDI Stream Callbacks.
- How to Generate a MIDI Stream Callback.
- Determining Resolution of Periodic MIDI Stream Callbacks.
- Processing MIDI Stream Callbacks.
- Behavior of MIDI Stream Callbacks in Real-time.

A general familiarity with MIDI stream APIs and especially the MIDIEVENT data structure is assumed. For an introduction to MIDI streams, see the "Stream Buffers" and "MIDIEVENT" topics in the Win32 SDK Multimedia documentation.

MORE INFORMATION
================

Uses of MIDI Stream Callbacks

MIDI stream callbacks can be used to inform the application of playback progress while a MIDI stream is being played. This allows the application to display a graphical indication of the progress of the music playback, or perform some other periodic task in synchronization with the music. MIDI stream callbacks can also be used to synchronize other non-periodic events as well. For example, a MIDI stream-based rendition of the 1812 Overture could include callback events to trigger the playback of a wave file of a cannon fusillade at appropriate times.

How to Generate a MIDI Stream Callback

An application specifies that a MIDI event within a MIDI stream be used to generate a callback by setting the MEVT_F_CALLBACK flag in the dwEvent parameter of the MIDIEVENT structure before the stream is sent to the MIDI device with midiStreamOut(). There are several kinds of short events and long events, and any of these may be used to generate a callback. The following is an example of a short MIDI event with callback:

```
    MIDIEVENT me;
    me.dwDeltaTime = 48;  // play 48 ticks after preceding event
```

```
    me.dwStreamID = 0;    // must be 0
    me.dwEvent = MEVT_F_SHORT | MEVT_F_CALLBACK | // flags
   (((DWORD)MEVT_SHORTMSG) << 24) | // event code
        (DWORD) 0x90 |                   // MIDI note-on status byte
        (((DWORD) 0x3C) << 8) |          // middle C
        (((DWORD) 0x7F) << 16);          // maximum key velocity
```

Note that the flags and the event code must all be coded into the high byte of dwEvent, but only the flags are defined in Mmsystem.h as DWORD values with bits set in the high byte. The event codes such as MEVT_SHORTMSG are defined as BYTE values, so it is necessary to left-shift them to the high byte position. Similarly, two of the three bytes describing the event data are shifted to their appropriate positions in the lower 24 bits of dwEvent.

If a callback is desired at a time for which there are no MIDI events in the stream, a special event can be added that does not play back MIDI data but causes a callback to occur by using the MEVT_NOP "no-op" code. In the following example, the MIDI data is not played back through the MIDI output device, but a pointer is sent to the application in a callback:

```
    MIDIEVENT me;
    me.dwDeltaTime = 48;
    me.dwStreamID = 0;
    me.dwEvent = MEVT_F_SHORT | MEVT_F_CALLBACK |
        (((DWORD)MEVT_NOP) << 24) |
        (DWORD) 0x90 |
        (((DWORD) 0x3C) << 8) |
        (((DWORD) 0x7F) << 16);
```

Simultaneous callback events are not meaningful and there are practical limitations to spacing callbacks close together in time. These limitations are described in detail later in this article.

Determining Resolution of Periodic MIDI Stream Callbacks

Periodic callbacks can be generated by tagging appropriate events with a callback, and in cases where there are no events at the required times, inserting MEVT_NOP events with callbacks, as discussed previously. Note that there is no API for generating periodic MIDI stream callbacks similar to the timeSetEvent API for generating periodic multimedia timer callbacks. Instead, each callback event must be individually specified by a MIDIEVENT data structure in the MIDI stream buffer.

You should choose a time period (dwDeltaTime) that provides sufficient resolution for your application's needs but does not impose too great a performance penalty. While the optimum period will depend on many factors, it is probably not realistic to try for a period shorter than something on the order of 25 milliseconds. Also, by allowing a tolerance on this period, you can tag existing events with a callback that are not exactly on time, but sufficiently close to being on time that it will not be noticed by the user. As a result, many fewer MEVT_NOP events with callbacks must be inserted, which will result in more efficient playback performance. The amount of tolerance to allow depends on a number of factors, but for a period of 25 milliseconds, a tolerance of 10 milliseconds would be a

reasonable value to try.

Although there isn't a set hard-and-fast limit on the minimum effective callback period, the 25 milliseconds figure reflects realistic design limitations. You don't want to burn processor time trying to be more accurate than the user can possibly distinguish. And equally so, you don't want to burn processor time at the cost of application performance. Because MIDI stream callbacks result in system overhead which could potentially impact MIDI playback performance, callbacks should not be scheduled more frequently than absolutely necessary.

Processing MIDI Stream Callbacks

For each MIDI stream callback event, a MOM_POSITIONCB message is sent to the application's callback procedure. Along with this message, the callback procedure receives a pointer to a MIDIHDR structure, the dwOffset member of which specifies the offset into the MIDI stream buffer in bytes for the callback event most recently processed. Using this offset, the application can access the event data associated with the callback event to learn which event has occurred and perform any necessary processing.

The callback function used to process MIDI stream callbacks is similar to other MIDI callback functions. The following is an example of a MIDI stream callback function:

```
void CALLBACK MidiStreamProc(
    HMIDIOUT hmo,     // handle of MIDI output device
    UINT uMsg,        // MIDI message
    DWORD dwInstance, // callback instance data
    DWORD dwParam1,   // address of MIDI header
    DWORD dwParam2)   // unused parameter
{
    LPMIDIHDR lpMIDIHeader;
    MIDIEVENT *lpMIDIEvent;

    if (!dwParam1)
        return;    // nothing to do

    switch (uMsg)
    {
        case MOM_POSITIONCB:
            // assign address of MIDIHDR
            lpMIDIHeader = (LPMIDIHDR)dwParam1;

            // get address of event data
            lpMIDIEvent = (MIDIEVENT *)&(lpMIDIHeader->
                lpData[lpMIDIHeader->dwOffset]);

            // do something here with event data

            break;

        // handle these messages if desired
        case MOM_OPEN:
```

```
        case MOM_DONE:
        case MOM_CLOSE:
            break;
    }
}
```

Behavior of MIDI Stream Callbacks in Real-time

Look at the MIDI stream callback as a request by the application to be called after a specific event in the stream is reached during playback. Once the callback is delivered, the event referred to by the callback is always the absolutely most recent event processed that has the callback flag set. If the callbacks are far apart and the system is running well, this will usually be the same event that triggered the callback, but it doesn't have to be. Stated another way, if another event with the callback flag set has been processed by the time the callback is delivered, then the event referred to by the callback will be the most recently processed callback event, not the event that originally triggered the callback.

The dwOffset parameter of the MIDIHDR received by the callback function tells the application how far playback has travelled in the stream at the time the callback is finally given to the application. For callback events that are scheduled to occur at the same time, the same offset referring to the most recently processed callback event will be received with each of the callbacks. Therefore, the use of simultaneous callback events serves no useful purpose and results in unnecessary overhead, so it should be avoided. Similarly, if callback events are not scheduled simultaneously but are scheduled too closely together, the callbacks will also refer to the most recently processed callback event, so this situation should also be avoided.

When a callback is received by the application, it should use the dwOffset data to decide what to do. If it is expecting a callback at a particular location but gets a callback for somewhere further along in the stream instead, it means that the application is not keeping up or that the callbacks are scheduled too closely together for the capabilities of the machine. The best thing to do in this case is to just throw the callback away because there will be another one in the queue that will be delivered just as soon as processing of the current callback is completed. This may add a little complexity to the logic required to process the callbacks but not to any great extent, and it has the benefit of degrading gracefully on machines that can't keep up.

Additional Notes

Note that MIDI stream support is currently available only on Windows 95. For a complete implementation of a MIDI playback sequencer written with the MIDI stream APIs, see the MIDIPLYR sample in the Windows 95 multimedia samples in the Win32 SDK or the MSTREAM sample in the Games SDK. Although these samples do not use MIDI stream callbacks, they both demonstrate use of MIDI streams in general.

Additional reference words: 4.00 MEVT_EVENTPARM MEVT_EVENTTYPE

KBCategory: kbmm kbprg kbhowto kbcode
KBSubcategory: MMMidi

How to Use NetUserAdd

PSS ID Number: Q140165

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, 4.0

SUMMARY

NetUserAdd allows a Windows NT application to add a user to a Windows NT workstation or server programmatically. Adding a user with this function seems easy enough, but there are a few issues that are not obvious.

MORE INFORMATION

If you would like to add users to a Windows NT workstation or server in your application, you must use NetUserAdd. Given a server name, level, and appropriate structure, a new user will be created. The following pointers may help you avoid problems:

1. As required of all Net* functions, submit all strings in UNICODE (wide characters). Use MultiByteToWideChar to convert standard ASCII strings into UNICODE strings. Don't forget to include the terminating NULL in string conversions.

2. Specify the server name as \\server. Be sure to include double backslashes, four backslashes if inside a C literal string.

3. Use the UF_SCRIPT required flag. Without it, you will get ERROR_INVALID_PARAMETER.

4. When using level 2, make sure your strings do not exceed the maximum lengths. Constants are defined for the maximum lengths. For example, the maximum lengths for params, comment, user comment, and full name is MAXCOMMENTSZ. Paths must fit within PATHLEN characters. UNC names for servers must fit in UNCLEN characters.

5. To add user accounts, always use NetUserAdd, not guest or administrator accounts. After the account is established, use NetUserSetGroups to add Administrator, Guest, or other groups to the new user account.

REFERENCES

Microsoft LAN Manager, A Programmer's Guide by Ralph Ryan.
Microsoft LAN Manager Programmer's Reference.
Win32 SDK Documentation.

Additional reference words: NETUSERADD LANMAN 3.10 3.50 3.51 4.00
KBCategory: kbnetwork kbprg kbhowto

KBSubcategory: NtwkMisc

How to Use PeekMessage() Correctly in Windows

PSS ID Number: Q74042

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In the Windows environment, many applications use a PeekMessage() loop to perform background processing. Such applications must allow the Windows system to enter an idle state when their background processing is complete. Otherwise, system performance, "idle-time" system processes such as paging optimizations, and power management on battery-powered systems will be adversely affected.

While an application is in a PeekMessage() loop, the Windows system cannot go idle. Therefore, an application should not remain in a PeekMessage() loop after its background processing has completed.

NOTE: The PeekMessage method described in this article is only needed if your application is a 32-bit application for Windows and is, for some reason, unable to create threads and perform background processing.

MORE INFORMATION

Many Windows-based applications use PeekMessage() to retrieve messages while they are in the middle of a long process, such as printing, repaginating, or recalculating, that must be done "in the background." PeekMessage() is used in these situations because, unlike GetMessage(), it does not wait for a message to be placed in the queue before it returns.

An application should not call PeekMessage() unless it has background processing to do between the calls to PeekMessage(). When an application is waiting for an input event, it should call GetMessage() or WaitMessage().

Remaining in a PeekMessage() loop when there is no background work causes system performance problems. A program in a PeekMessage() loop continues to be rescheduled by the Windows scheduler, consuming CPU time and taking time away from other processes.

In enhanced mode, the virtual machine (VM) in which Windows is running will not appear to be idle as long as an application is calling the PeekMessage function. Therefore, the Windows VM will continue to receive a considerable fraction of CPU time.

Many power management methods employed on laptop and notebook computers are based on the system going idle when there is no processing to do. An application that remains in a PeekMessage() loop will make the system appear busy to power management software, resulting in excessive power consumption and shortening the time that the user can run the system.

In the future, the Windows system will make more and more use of idle time to do background processing, which is designed to optimize system performance. Applications that do not allow the system to go idle will adversely affect the performance of these techniques.

All these problems can be avoided by calling the PeekMessage() function only when there is background work to do, and calling GetMessage() or WaitMessage() when there is no background work to do.

For example, consider the following PeekMessage() loop. If there is no background processing to do, this loop will continue to run without waiting for messages, preventing the system from going idle and causing the negative effects described above.

```
   // This PeekMessage loop will NOT let the system go idle.

   for( ;; )
   {
      while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
         if (msg.message == WM_QUIT)
            return TRUE;

         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }

      BackgroundProcessing();
    }
```

This loop can be rewritten in two ways, as shown below. Both of the following PeekMessage() loops have two desirable properties:

- They process all input messages before performing background processing, providing good response to user input.

- The application "idles" (waits for an input message) when no background processing needs to be done.

Improved PeekMessage Loop 1

```
   // Improved PeekMessage() loop

   for(;;)
   {
      while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
      {
         if (msg.message == WM_QUIT)
```

```
            return TRUE;

         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }

      if (IfBackgroundProcessingRequired())
         BackgroundProcessing();
      else
         WaitMessage(); // Will not return until a message is posted.
   }
```

Improved PeekMessage Loop 2

```
   // Another improved PeekMessage() loop

   for (;;)
   {
      for (;;)
      {
         if (IfBackgroundProcessingRequired())
         {
            if (!PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
               break;
         }
         else
            GetMessage(&msg, NULL, 0, 0, 0);

         if (msg.message == WM_QUIT)
            return TRUE;

         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }
      BackgroundProcessing();
   }
```

Note that calls to functions such as IsDialogMessage() and TranslateAccelerator() can be added to these loops as appropriate.

There is one case in which the loops above need additional support: if the application waits for input from a device (for example, a fax board) that does not send standard Windows messages. For the reasons outlined above, a Windows-based application should not use a PeekMessage() loop to continuously poll the device. Rather, implement an interrupt service routine (ISR) in a dynamic-link library (DLL). When the ISR is called, the DLL can use the PostMessage function to inform the application that the device requires service. DLL functions can safely call the PostMessage() function because the PostMessage() function is reentrant.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrMsg

How to Use RPC Under Win32s

PSS ID Number: Q127903

The information in this article applies to:

- Microsoft Win32s, versions 1.2 and 1.25a

Win32s does not thunk Remote Procedure Call (RPC) calls. Therefore, if you have a Win32-based application that uses RPC and you want it to run on Win32s, you will need to write a thunking layer for your application to thunk the 32-bit calls to the 16-bit RPC implementation.

One issue to keep in mind when writing the thunking layer is how to handle stub code. It is convenient to direct the MIDL compiler to produce 16-bit stub code for the application. The stub code can be built as a 16-bit DLL, which can be called from the Win32-based application via the Universal Thunk. This eliminates the need to write a thunking layer for the RPC run-time functions that appear in 32-bit client stub code.

One limitation is that RPC servers are not supported with the 16-bit RPC, only clients. Microsoft's 32-bit RPC implements servers by spawning threads for each call, and threads are not supported under Win32s. There may be other vendors who supply a 16-bit RPC that supports servers.

Additional reference words: 1.20
KBCategory: kbprg
KBSubcategory: W32s

How to Use SS_GRAYRECT SS_BLACKRECT SS_WHITERECT in Windows 95

PSS ID Number: Q125684

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The colors used in the gray, white, and black rectangle static controls has changed in Windows 95. In previous versions of Windows, these colors were based on the system colors for windows. In Windows 95, these colors are based on the colors for 3D objects.

MORE INFORMATION
================

In Windows 95, the definitions for white, gray, and black rectangle static controls are as follows:

- SS_WHITERECT: Specifies a rectangle filled with the highlight color for three-dimensional display elements (for edges facing the light source). This is the same color retrieved by using GetSysColor() with COLOR_3DHILIGHT.

- SS_GRAYRECT: Specifies a rectangle filled with the shadow color for three-dimensional display elements (for edges facing away from the light source). This is the same color retrieved by using GetSysColor() with COLOR_3DSHADOW.

- SS_BLACKRECT: Specifies a rectangle filled with the Shadow color for three-dimensional display elements (for edges facing away from the light source). This is the same color retrieved by using GetSysColor() with COLOR_3DDKSHADOW. This is not the same color as COLOR_3DSHADOW. There are two shadow colors used on 3D objects.

In previous versions of Windows, the definitions for white, gray, and black rectangle static controls were as follows:

- SS_WHITERECT: Specifies a rectangle filled with the color used to fill window backgrounds. This color is white in the default Windows color scheme. This is the same color retrieved by using GetSysColor() with COLOR_WINDOW.

- SS_GRAYRECT: Specifies a rectangle filled with the color used to fill the screen background. This color is gray in the default Windows color scheme. This is the same color retrieved by using GetSysColor() with COLOR_BACKGROUND.

- SS_BLACKRECT: Specifies a rectangle filled with the color used to draw

window frames. This color is black in the default Windows color scheme. This is the same color retrieved by using GetSysColor() with COLOR_WINDOWFRAME.

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrCtl

How to Use the AutoDial Feature in Network and WinSock Apps

PSS ID Number: Q137369

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SUMMARY
=======

This article describes how to use the AutoDial feature in applications using Windows sockets.

MORE INFORMATION
================

Windows 95 supports a new AutoDial feature. Using this feature, applications can initiate a dial-up connection to remote networks as needed. AutoDial is fully transparent from the programmer's point of view.

The following steps demonstrate this capability:

1. Ping abc.abc.abc.abc

2. If the destination IP address (or fully qualified name) cannot be resolved or connected locally (on a LAN for example), then the AutoDial window comes up, and a preconfigured Dial-Up Network Connection connects to a WAN link provider.

These steps also apply to any Windows Socket application that requests a connection to a destination that cannot be connected locally. There are no special coding techniques needed to program such applications.

The following versions of Window 95 have the AutoDial feature available:

- OEM preinstalled Windows 95
- Windows 95 with the Plus! pack.

The retail version of Windows 95 does not have the AutoDial feature. To use the AutoDial feature on the retail version of Windows 95, you must install the Microsoft Internet Explorer, which is available free of charge on the Microsoft Web site (http://www.microsoft.com) subject to the licensing agreement. Once Microsoft Internet Explorer is installed, the Internet icon appears in the control panel. AutoDial can be configured by double-clicking this icon.

At the present time AutoDial is not available on Windows NT.

Additional reference words: 4.00
KBCategory: kbnetwork
KBSubcategory: NtwkWinsock

How to Use the Small Icon in Windows 95

PSS ID Number: Q125682

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SUMMARY

In Windows 95, each application is associated with two icons: a small icon (16x16) and a large icon (32x32). The small icon is displayed in the upper-left hand corner of the application and on the taskbar.

MORE INFORMATION

Large and small icons are associated with an application by using the RegisterClassEx() function. This function takes a pointer to a WNDCLASSEX structure. The WNDCLASSEX structure is similar to the WNDCLASS structure except for the addition of the hIconSm parameter, which is used for the handle to the small icon. If no small icon is associated with an application, Windows 95 will use a 16x16 representation of the large icon.

NOTE: RegisterClassEx() is not currently implemented in Windows NT where it returns NULL.

The LoadIcon() function loads the large icon member of an icon resource. To load the small icon, use the new LoadImage() function as follows:

```
LoadImage(  hInstance,
            MAKEINTRESOURCE(<icon identifier>),
            IMAGE_ICON,
            16,
            16,
            0);
```

The small icon currently associated with the application will be displayed in the upper-left corner of the application's main window and on the task bar. Both the large and the small icon association can be changed at runtime by using the WM_SETICON message.

By default, the start menu will display the first icon defined in an application's resources. This can be changed through the start menu property sheets.

Explorer displays the first defined icon in an application's resources unless the application adds an entry to the registry under the program information called DefaultIcon or defines an icon handler shell extension for the file type. Refer to the Shell Extension documentation for more information on shell extensions.

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrWndw

How to Use the StartService API Within a Service

PSS ID Number: Q133756

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51

SUMMARY

=======

In the Win32 API Help file, it mentions that you cannot call the StartService API from within a service. This statement is not totally correct. It should say that you cannot call the StartService API from within a service while it is initializing.

An initializing service is a service which has not yet reported to the Service Control Manager that it has successfully started. Once the service is running, you can use the StartService API within a service.

MORE INFORMATION

================

If your service is initializing due to a dependency or the auto start flag, and your service attempts to start another service, a deadlock state in the Service Control Manager may occur. The reason this occurs is because the call to StartService() blocks because the Service Control Manager has a lock on the service control database from the original initializing service.

Additional reference words: 3.50

KBCategory: kbprg kbdocerr

KBSubcategory: BseService

How to Use the Win32 registry from 16-bit Windows-Based App

PSS ID Number: Q137378

The information in this article applies to:

 - Microsoft Win32 Software Development KIT (SDK)
   versions 3.1x, 3.5, 3.51, 4.0

SUMMARY
=======

By obtaining access to all Win32-based registry APIs, a 16-bit Windows-based application can also gain full access to the Win32 registry. Such access would be based on Generic Thunking to ensure portability between Windows NT and Windows 95.

MORE INFORMATION
=================

A 16-bit Windows-based application must implement a generic thunk function for each Win32 API required to access the Win32 registry. To achieve the greatest flexibility, you need to implement these new functions in a 16-bit Windows-based DLL that can be implicitly linked to a 16-bit Windows-based application. In Windows NT and Windows 95, all Win32 registry APIs are exported from Advapi.dll.

For a 16-bit Windows-based application to access the Win32 registry, it must first define the predefined reserved handle value (HKEY_LOCAL_MACHINE) as defined by the Win32 environment. These predefined reserved handle values are located in the Winreg.h file in the Win32 SDK.

The following demonstrates a sample technique used to perform registry thunking.

```
// required includes
#include "wownt16.h"     // available from Win32 SDK
#include "shellapi.h"    // 16-bit Windows header

// As defined by the Win32 SDK in Winreg.h
#define HKEY_CURRENT_USER           (( HKEY ) 0x80000001 )

HKEY hOpen;
DWORD rc;

if ( ERROR_SUCCESS ==
      (rc=RegOpenKey(HKEY_CURRENT_USER, "Control Panel\\Desktop", &hOpen)))
      {
         DWORD dwType, cbData;
         BYTE   bData[1000];
         cbData = 1000;

         if ( (rc =
         RegQueryValueEx(hOpen, "BorderWidth", NULL,
```

```
                              &dwType, bData, &cbData ))
                == ERROR_SUCCESS )
            {
            // used retreived data
            MessageBox(NULL, "Value Retreived","Information",MB_OK);
            }
            else
               MessageBox(NULL, "RegQueryValueEx Failed","Error",MB_OK);
               RegCloseKey(hOpen);
      }

// This code should be placed in a DLL to allow for maximum flexibility.
// When the full conversion to 32-bit is complete, the thunking library
// need not be included.

LONG RegQueryValueEx(
    HKEY  hKey,
    LPSTR  lpszValueName,
    LPDWORD  lpdwReserved,
    LPDWORD  lpdwType,
    LPBYTE  lpbData,
    LPDWORD  lpcbData
   )
{
    DWORD pFn;
    DWORD dwResult = ERROR_ACCESS_DENIED; //random error if fail
    DWORD hAdvApi32;

    hAdvApi32=LoadLibraryEx32W("ADVAPI32.DLL", NULL, 0);

    if ((DWORD)0!=hAdvApi32)
     {
           // call ANSI version
      pFn=GetProcAddress32W(hAdvApi32, "RegQueryValueExA");
      if ((DWORD)0!=pFn)
       {
        dwResult=CallProcEx32W(
               CPEX_DEST_STDCALL | 6, // standard function call with
                                      // six parameters
               0x3e,                  // Identify what parameters
                                      //(addresses) must be translated
               pFn,                   // function pointer
               hKey,                  // open key
               lpszValueName,         // value to query
               lpdwReserved,          // reserved for future use
               lpdwType,              // value type
               lpbData,               // value data
               lpcbData               // value data length
               );
       }
     }
    if (hAdvApi32)
                FreeLibrary32W(hAdvApi32);

              return(dwResult);
```

```
}

To include the generic thunk APIs, the 16-bit app needs to add the
following to its module definition (.DEF) file:

IMPORTS
        _CallProcEx32W          = KERNEL.518
        LoadLibraryEx32W        = KERNEL.513
        FreeLibrary32W          = KERNEL.514
        GetProcAddress32W       = KERNEL.515

Additional reference words: 3.50 4.00
KBCategory: kbprg kbcode
KBSubcategory: Bse
```

How to Use Wave Audio Volume Control APIs

PSS ID Number: Q139098

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, 4.0

SUMMARY

=======

The functions waveOutGetVolume() and waveOutSetVolume() can be used to set and retrieve a wave output device's volume. The DWORD passed or retrieved contains the left channel's volume in the low-order word and the right channel's volume in the high-order word. In this article, the use of these functions is illustrated through a single function that can be added to the Generic sample.

MORE INFORMATION

================

Although the Mixer APIs included in the Win32 SDK offer more extensive control, quick and simple audio volume control can be obtained by using the waveOutGetVolume() and waveOutSetVolume() functions.

By adding the code for the ShowVolume() function (given in this article) to the Generic sample, you can cause it to output message boxes about the state of the volume control. While the sample is running, the WSS Audio Conrol's Volume Control (or a given sound card's mixer control that displays wave output volume) undergoes changes in its slider positions as the values for right and left channel volume are changed by the ShowVolume() function.

As the sample runs, the first message box reports the number of wave output devices. The second and third message boxes report a left channel volume of 8192 and a right channel volume of 16384 (out of a maximum of 65535). The horizontal slider in the Wave column of the Volume Control is the balance slider, and it is set to the right because the right volume is greater now. Also, the vertical slider in the Wave column, which is the volume slider, is set to the 1/4 position, tracking the larger volume setting of the two channels.

The fourth and fifth messages boxes report a left and right volume of 32768 and 16384 respectively. Now the balance slider is to the left, and the wave volume is about 1/2, again tracking the larger value.

ShowVolume() Code to Be Added to the Generic Sample

```
// Link with mmsystem.lib for 16-bit applications and
// link with winmm.lib for 32-bit applications.
```

```
#include <mmsystem.h>
#include <stdlib.h>

void ShowVolume(void);   //Prototype the function early in the app

void ShowVolume(void)
{
    // This is the function that can be added to the Generic Sample to
    // illustrate the use of waveOutGetVolume() and waveOutSetVolume().

    char buffer[40];
    char printbuf[80];
    UINT uRetVal, uNumDevs;
    DWORD volume;
    long lLeftVol, lRightVol;

    WAVEOUTCAPS waveCaps;

    // Make sure there is at least one
    // wave output device to work with.
    if (uNumDevs = waveOutGetNumDevs())
    {
   itoa((int)uNumDevs, buffer, 10);
   wsprintf(printbuf, "Number of devices is %s\n", (LPSTR)buffer);
   MessageBox(GetFocus(), printbuf, "NumDevs", MB_OK);
    }

    // This sample uses a hard-coded 0 as the device ID, but retail
    // applications should loop on devices 0 through N-1, where N is the
    // number of devices returned by waveOutGetNumDevs().
    if (!waveOutGetDevCaps(0,(LPWAVEOUTCAPS)&waveCaps,
       sizeof(WAVEOUTCAPS)))

    {
   // Verify the device supports volume changes
   if(waveCaps.dwSupport & WAVECAPS_VOLUME)
   {
       // The low word is the left volume, the high word is the right.
       // Set left channel: 2000h is one-eighth volume (8192 base ten).
       // Set right channel: 4000h is quarter volume (16384 base ten).
       uRetVal = waveOutSetVolume(0, (DWORD)0x40002000UL);

       // Now get and display the volumes.
       uRetVal = waveOutGetVolume(0, (LPDWORD)&volume);

       lLeftVol = (long)LOWORD(volume);
       lRightVol = (long)HIWORD(volume);

       ltoa(lLeftVol, buffer, 10);
       wsprintf(printbuf, "Left Volume is %s\n", (LPSTR)buffer);
       MessageBox(GetFocus(), printbuf, "Left Volume", MB_OK);

       ltoa(lRightVol, buffer, 10);
       wsprintf(printbuf, "Right Volume is %s\n", (LPSTR)buffer);
       MessageBox(GetFocus(), printbuf, "Right Volume", MB_OK);
```

```
        // The low word is the left volume, the high word is the right.
        // Set left channel: 8000h is half volume (32768 base ten).
        // Set right channel: 4000h is quarter volume (16384 base ten).
        uRetVal = waveOutSetVolume(0, (DWORD)0x40008000UL);

        // Now get and display the volumes.
        uRetVal = waveOutGetVolume(0, (LPDWORD)&volume);

        lLeftVol = (long)LOWORD(volume);
        lRightVol = (long)HIWORD(volume);

        ltoa(lLeftVol, buffer, 10);
        wsprintf(printbuf, "Left Volume is %s\n", (LPSTR)buffer);
        MessageBox(GetFocus(), printbuf, "Left Volume", MB_OK);

        ltoa(lRightVol, buffer, 10);
        wsprintf(printbuf, "Right Volume is %s\n", (LPSTR)buffer);
        MessageBox(GetFocus(), printbuf, "Right Volume", MB_OK);

    }
     }
}
```

Additional reference words: 3.50 4.00 waveOutGetVolume waveOutSetVolume
KBCategory: kbmm kbsound kbprg kpcode kbhowto
KBSubcategory: MMWave

How to Use Win32 API to Draw a Dragging Rectangle on Screen DC

PSS ID Number: Q135865

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

This article shows by example how to have your application draw a dragging rectangle on an arbitrary area of the screen. This action is known as "rubber banding." Utilities that perform screen capture, zooming, and so on are typical examples of such applications. Due to the change in behavior of the SetCapture API in win32, the standard technique used for rubber banding on the screen DC in 16-bit Windows no longer works under Windows NT and Windows95. This article explores alternate methods you can use to perform rubber banding in win32.

MORE INFORMATION

16-bit Method for Rubber Banding Won't Work in Win32-Based Applications

In 16-bit Windows, an application called the SetCapture API prior to starting the rubber banding. This then directed all further mouse input to the application's capture window. This allowed the user to click anywhere on the desktop including the client area of other running applications and start the drag action. Because SetCapture directs all mouse input to the capture window, it is then a simple matter for the application to draw a rubber banding rectangle as the user moves the mouse.

However, the Win32 implementation of the SetCapture API has changed so that the 16-bit method fails. SetCapture no longer directs all mouse input to an application's capture window. If an application calls SetCapture and the user then clicks another application or the desktop (an application by itself) to activate it, subsequent mouse messages are no longer directed to the capture window. The only mouse messages that a capture window receives are those generated when the application's thread is active. This is applicable to a 16-bit application running under Windows NT or Windows 95 as well as a newly written 32-bit application. There are two ways for a Win32-based application to perform rubber banding.

Method One: How to Perform Rubber Banding in Win32-Based Applications

In this method, the application imposes a restriction on the user as to how the dragging action is done. For example, the application might display a small window and require the user to depress the mouse button in the client

area of the window to indicate the beginning of the rubber banding mode and drag the mouse, without releasing the mouse button, to whichever part of the desktop the user wants to capture. Upon reaching the appropriate anchor point, the user presses a key on the keyboard (for example, the CTRL key) and drags the mouse to perform the actual rubber banding. Note that during the entire process, the user keeps the mouse button depressed, so other applications are not activated, and all the mouse messages are therefore directed to the capture window.

The following sample code illustrates how this might be implemented:

```
// some global declarations

BOOL bDrag;
BOOL bDraw;
POINT ptCurrent, ptNew;

// inside the capture window procedure...

case WM_LBUTTONDOWN:
   SetCapture(hwnd);
   bDrag = TRUE;   // set flag to check for key down in mouse move message

case WM_MOUSEMOVE:
   if (bDrag)
   {
      if ( keyFlags & MK_CONTROL )  // CTRL key down
      {
         bDraw = TRUE;  // start actual drawing from next move message
         ptCurrent.x = ptNew.x = x;  // store the first point after
         ptCurrent.y = ptNew.y = y;  // converting to screen coordinates
         ClientToScreen (hwnd,&ptCurrent);
         ClientToScreen (hwnd,&ptNew);
         bDrag = FALSE;
      }
   }
   else if ( bDraw )
   {

      // Draw two rectangles in the screen DC to cause rubber banding

      HDC hdc = CreateDC ( TEXT("DISPLAY"), NULL, NULL, NULL );
      SetROP2(hdc, R2_NOTXORPEN );

      // Draw over and erase the old rectangle

      Rectangle   ( hdc, ptCurrent.x, ptCurrent.y, ptNew.x, ptNew.y );
      ptNew.x = x;
      ptNew.y = y;
      ClientToScreen (hwnd,&ptNew);

      // Draw the new rectangle

      Rectangle   ( hdc, ptCurrent.x, ptCurrent.y, ptNew.x, ptNew.y );
      DeleteDC ( hdc );
```

```
   }

case WM_LBUTTONUP:

   if ( bDraw )
   {

      // Don't leave orphaned rectangle on desktop; erase last rectangle.

      HDC hdc;
      bDrag = FALSE;
      bDraw = FALSE;
      ReleaseCapture();
      hdc = CreateDC ( TEXT("DISPLAY"), NULL, NULL, NULL );
      SetROP2(hdc, R2_NOTXORPEN );
      Rectangle   ( hdc, ptCurrent.x, ptCurrent.y, ptNew.x, ptNew.y );
      DeleteDC ( hdc );

      // At this point the application has the coordinates of the rubber
      // banding rectangle.

   } else if ( bDrag)
   {
      // The user released the mouse button without ever pressing CTRL key.

      bDraw = FALSE;
      bDrag = FALSE;
      ReleaseCapture();
   }
```

The drawback to this approach is that the rubber banding action may not be intuitive. The user is required to use a key in addition to the mouse now, whereas in 16-bit Windows this could have been accomplished with the mouse alone.

Method Two: How to Perform Rubber Banding in Win32-Based Applications

Another way to do rubber banding in win32 consistent with the 16-bit method is to create a popup window that covers the entire desktop and has the WS_EX_TRANSPARENT style bit set. This window can be created with no title bar or borders and is active only for the time period of the rubber banding. It is destroyed upon completion of the rubber banding - when the user releases the mouse button. This popup window becomes the capture window and the rubber banding logic would then simply be applied on the client area DC of the popup window.

Additional reference words: 4.00
KBCategory: kbui kbcode
KBSubcategory: usrctl

How to Use WinSock to Enumerate Addresses

PSS ID Number: Q129315

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.5 and 3.51

SUMMARY

The gethostbyname() and gethostname() WinSock database APIs can be used to list IP addresses for a multihomed host. However, these functions work only for IP addresses. This article shows by example how to give addresses for other address families. Two different methods are given.

MORE INFORMATION

Method One Code Sample

AF_IPX:

This function can be used to give an IPX address:

```
   #include <winsock.h>
   #include <wsipx.h>
   #include <wsnwlink.h>

   #include <stdlib.h>

   // Note:  In the interest of clarity, the following code does not check
   //        return values or handle error conditions.

   void IPXEnum()
   {
      int            cAdapters,
                     cbOpt  = sizeof( cAdapters ),
                     cbAddr = sizeof( SOCKADDR_IPX );

      SOCKET         s;
      SOCKADDR_IPX   Addr;

      // Create IPX socket.
      s = socket( AF_IPX, SOCK_DGRAM, NSPROTO_IPX );

      // Socket must be bound prior to calling IPX_MAX_ADAPTER_NUM
      memset( &Addr, 0, sizeof( Addr ));
      Addr.sa_family = AF_IPX;
      bind( s, (SOCKADDR*) &Addr, cbAddr);

      // Get the number of adapters => cAdapters.
```

```
      getsockopt( (SOCKET) s, NSPROTO_IPX, IPX_MAX_ADAPTER_NUM,
                  (char *) &cAdapters, &cbOpt );
      // At this point cAdapters is the number of installed adapters.
      while ( cAdapters > 0 )
      {
         IPX_ADDRESS_DATA  IpxData;

         memset( &IpxData, 0, sizeof(IpxData));

         // Specify which adapter to check.
         IpxData.adapternum = cAdapters - 1;
         cbOpt = sizeof( IpxData );

         // Get information for the current adapter.
         getsockopt( s, NSPROTO_IPX, IPX_ADDRESS,
                     (char*) &IpxData, &cbOpt );

         // IpxData contains the address for the current adapter.
         cAdapters--;
      }
   }
```

AF_NETBIOS:

This function uses the EnumProtocols() function to give lana numbers for the available NetBIOS transports. NOTE: This doesn't work under Windows NT 3.5 because of a bug in EnumProtocols(), but it does work under Windows NT 3.51.

```
   void NBEnum()
   {
      DWORD          cb = 0;
      PROTOCOL_INFO *pPI;
      BOOL           pfLanas[100];

      int            iRes,
                     nLanas = sizeof(pfLanas) / sizeof(BOOL);

      // Specify NULL for lpiProtocols to enumerate all protocols.

      // First, determine the output buffer size.
      iRes = EnumProtocols( NULL, NULL, &cb );

      // Verify the expected error was received.
      assert( iRes == -1 && GetLastError() == ERROR_INSUFFICIENT_BUFFER );
      if (!cb)
      {
         fprintf( stderr, "No available NetBIOS transports.\n");
         break;
      }

      // Allocate a buffer of the specified size.
      pPI = (PROTOCOL_INFO*) malloc( cb );

      // Enumerate all protocols.
```

```
      iRes = EnumProtocols( NULL, pPI, &cb );

      // EnumProtocols() lists each lana number twice, once for
      // SOCK_DGRAM and once for SOCK_SEQPACKET. Set a flag in pfLanas
      // so unique lanas can be identified.

      memset( pfLanas, 0, sizeof( pfLanas ));

      while (iRes > 0)
         // Scan protocols looking for AF_NETBIOS.
         if ( pPI[--iRes].iAddressFamily == AF_NETBIOS )
            // found one
            pfLanas[ pPI[iRes].iProtocol ] = TRUE;

      fprintf( stderr, "Available NetBIOS lana numbers: " );
      while( nLanas-- )
         if ( pfLanas[nLanas] )
            fprintf( stderr, "%d ", nLanas );

      free( pPI );
   }
```

AF_APPLETALK:

Address enumeration is not meaningful for AF_APPLETALK. On a multihomed host with routing disabled, only the default adapter is used. If routing is enabled, a single AppleTalk address is used for all installed network adapters.

Method Two: Code Sample

Listed below is an example of how to use the WinSock database APIs to give IP addresses:

```
   void EnumIP()
   {
      char     szHostname[100];
      HOSTENT *pHostEnt;
      int      nAdapter = 0;

      gethostname( szHostname, sizeof( szHostname ));
      pHostEnt = gethostbyname( szHostname );

      while ( pHostEnt->h_addr_list[nAdapter] )
      {
         // pHostEnt->h_addr_list[nAdapter] is the current address in host
         //  order.

         nAdapter++;
      }
   }
```

Additional reference words: 3.50
KBCategory: kbnetwork kbcode

KBSubcategory: NtwkWinsock

How Win32 Applications Can Read CD-ROM Sectors in Windows 95

PSS ID Number: Q137813

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with: Microsoft Windows 95 version 4.0

SUMMARY

=======

Some Win32 applications, such as multimedia applications and games, need to read sectors directly from compact discs to implement custom access or read caching to optimize CD-ROM access for specific purposes. This article provides information and example code that demonstrates how Win32-based applications can read sectors from compact discs in Windows 95.

MORE INFORMATION

================

The Windows 95 Compact Disc File System (CDFS) is a protected-mode CD-ROM driver that implements the V86-mode MSCDEX Interrupt 2Fh functions as its sole application-callable interface. This interface allows it to replace real-mode CD-ROM drivers and MSCDEX.EXE and still be completely compatible with existing applications for MS-DOS and Windows 3.x. In fact, applications for MS-DOS and Windows 3.x will not be able to tell the difference between CDFS and MSCDEX.EXE.

Windows 95 provides an interface for Win32-based applications to read sectors from compact discs; this interface differs substantially from that provided by Windows NT. Unlike Windows NT, which uses CreateFile() and ReadFile() to read sectors from compact discs, Windows 95 does not provide a Win32 API function for reading sectors. Instead, Win32-based applications running under Windows 95 must use an indirect method that involves thunking to a 16-bit DLL that calls MSCDEX functions.

The MSCDEX interface that CDFS provides is callable only in V86-mode, so Win32-based applications must thunk to a 16-bit DLL, and the 16-bit DLL must use the DOS Protected Mode Interface (DPMI) Simulate Real Mode Interrupt function to call MSCDEX Interrupt 2Fh functions. The Win32-based application must pass a buffer to the 16-bit DLL by using the thunk. Inside the thunk, the 16-bit DLL obtains the sector data from MSCDEX, copies it into the Win32-based application's buffer, and returns to the Win32-based application.

The following code demonstrates how to read sectors from a compact disc using the method described above. Because the code implements a flat thunk, it is divided into three sections:

- Code that resides in a Win32 DLL (the 32-bit side of the thunk).

- Code that resides in the thunk script.

```
 - Code that resides in the 16-bit DLL that actually calls MSCDEX.

//--------------------------------------------------------------------
// Code inside the Win32 DLL (the 32-bit side of the thunk)
// ReadSectorsFromCD() is exported by the DLL so that it can be called
// by Win32-based applications.

#include <windows.h>

#define CD_SECTOR_SIZE         2048

__declspec(dllexport)
BOOL WINAPI ReadSectorsFromCD (BYTE   bDrive,
                               DWORD  dwStartSector,
                               WORD   wSectors,
                               LPBYTE lpBuff);

// Prototype for thunk function in 16-bit DLL.
BOOL FAR PASCAL ReadCDRomSectors (BYTE   bDrive,
                                  DWORD  dwStartSector,
                                  WORD   wSectors,
                                  LPBYTE lpBuffer);


/*--------------------------------------------------------------------
  ReadSectorsFromCD()

  Calls the thunked function, ReadCDRomSectors(). This function is
  exported by the Win32 DLL and thus is callable from Win32-based
  applications.

  Parameters:

     bDrive
        Drive letter of CD-ROM drive to read from. Specified as
        a character in the range 'A', 'B', 'C', ..., 'Z'.  May be
        upper- or lower-case.

     dwStartSector
        First sector to read.

     wSectors
        Number of sectors to read.

     lpBuffer
        Buffer to contain sector data. Must be large enough to
        accomodate all sectors being read. Because this buffer is
        provided to the 16-bit DLL via a thunk, its maximum
        length is limited to 64K.

  Example use:
     // Read 5 sectors from CD-ROM drive E: starting at sector 16.
     fResult = ReadSectorsFromCD ('E', 16, 5, lpBuff);

  Return Value
```

```
     Returns TRUE if successful, or FALSE if an error occurred
     in reading or if a parameter was invalid.
-------------------------------------------------------------------*/

__declspec(dllexport)
BOOL WINAPI ReadSectorsFromCD (BYTE   bDrive,
                               DWORD  dwStartSector,
                               WORD   wSectors,
                               LPBYTE lpBuff)
{
   // Call 16-bit DLL to read sectors via a 32->16 thunk
   return ReadCDRomSectors (bDrive, dwStartSector, wSectors, lpBuff);
}


//--------------------------------------------------------------------
// Contents of the thunk script

enablemapdirect3216 = true;

typedef unsigned short WORD;
typedef unsigned long  DWORD;
typedef unsigned char  BYTE, *LPBYTE;
typedef bool           BOOL;

BOOL ReadCDRomSectors (BYTE   bDrive,
                       DWORD  dwStartSector,
                       WORD   wSectors,
                       LPBYTE lpBuffer)
{
   lpBuffer = inout;
}


//--------------------------------------------------------------------
// Code inside the 16-bit DLL. This code implements the 16-bit side of
// the thunk, and is where the calls to MSCDEX are made to read
// sectors from a compact disc.

#include <windows.h>
#include <string.h>
#include <ctype.h>

// Cooked-mode sector size in bytes
#define CD_SECTOR_SIZE         2048

// Maximum sector buffer size in bytes
#define MAX_BUFFER_LENGTH      65536

// Maximum muber of sectors for each read request
#define MAX_CD_SECTORS_TO_READ ((MAX_BUFFER_LENGTH) / \
                                (CD_SECTOR_SIZE))

// Processor flags masks -- use for testing wFlags member of RMCS
#define CARRY_FLAG  0x0001
```

```
// Real-mode call structure for making DPMI Simulate Real Mode
// Interrupt calls.
typedef struct tagRMCS
{
   DWORD edi, esi, ebp, RESERVED, ebx, edx, ecx, eax;
   WORD  wFlags, es, ds, fs, gs, ip, cs, sp, ss;
} RMCS, FAR* LPRMCS;

BOOL FAR PASCAL __export ReadCDRomSectors (BYTE   bDrive,
                                           DWORD  dwStartSector,
                                           WORD   wSectors,
                                           LPBYTE lpBuffer);

BOOL FAR PASCAL MSCDEX_ReadSector (BYTE   bDrive,
                                   DWORD  dwStartSector,
                                   LPBYTE RMlpBuffer);

BOOL FAR PASCAL SimulateRM_Int (BYTE bIntNum, LPRMCS lpCallStruct);
void FAR PASCAL BuildRMCS (LPRMCS lpCallStruct);

/*--------------------------------------------------------------------
  ReadCDRomSectors()

  Reads a specified number of sectors from a CD-ROM.

  Parameters:

     bDrive
        Drive letter of CD-ROM drive to read from. Specified as
        a character in the range 'A', 'B', 'C', ..., 'Z'.  May be
        upper- or lower-case.

     dwStartSector
        First sector to read.

     wSectors
        Number of sectors to read.

     lpBuffer
        Buffer to contain sector data. Must be large enough to
        accomodate all sectors being read. Since this buffer is
        provided by the calling Win32 application, no more than
        the first 64K bytes will be used.

  Return Value
     Returns TRUE if successful, or FALSE if an error occurred
     in reading or if a parameter was invalid.
--------------------------------------------------------------------*/

BOOL FAR PASCAL __export ReadCDRomSectors (BYTE   bDrive,
                                           DWORD  dwStartSector,
                                           WORD   wSectors,
                                           LPBYTE lpBuffer)
{
```

```
BOOL   fResult;
DWORD  cbOffset;
DWORD  i;
DWORD  gdaBuffer;     // Return value of GlobalDosAlloc().
LPBYTE RMlpBuffer;    // Real-mode buffer pointer
LPBYTE PMlpBuffer;    // Protected-mode buffer pointer


// Convert drive letter into drive number for MSCDEX call.
bDrive = toupper(bDrive) - 'A';

/*
   Validate parameters:
      bDrive must be between 0 and 25, inclusive.
      lpBuffer must not be NULL.
      wSectors must be between 1 and the maximum number of
         sectors that can fit into a 64K buffer, inclusive.
*/
if (bDrive > 25 || !lpBuffer)
   return FALSE;

if (!wSectors || (wSectors > MAX_CD_SECTORS_TO_READ))
   return FALSE;

/*
   Allocate buffer for MSCDEX call. This buffer must be below 1MB
   because the MSCDEX function will be called using DPMI. Like real-
   mode MSCDEX.EXE, CDFS implements the MSCDEX API as V86-mode
   Interrupt 2Fh functions.

   Free memory below 1MB is relatively scarce, so allocating a
   small sector buffer increases the chances that it can be
   allocated no matter how many other applications and MS-DOS
   device drivers are running. Also, a small sector buffer leaves
   more memory for other applications to use.
*/
gdaBuffer = GlobalDosAlloc (CD_SECTOR_SIZE);

if (!gdaBuffer)
   return FALSE;

RMlpBuffer = (LPBYTE)MAKELONG(0, HIWORD(gdaBuffer));
PMlpBuffer = (LPBYTE)MAKELONG(0, LOWORD(gdaBuffer));

// Call MSCDEX to read each sector.
for (i = cbOffset = 0;
     i < wSectors;
     i++, cbOffset += CD_SECTOR_SIZE)
   {
   if (fResult = MSCDEX_ReadSector (bDrive,
                                    dwStartSector + i,
                                    RMlpBuffer))
      _fmemcpy (lpBuffer + cbOffset, PMlpBuffer, CD_SECTOR_SIZE);
   else
      break;
```

```
      }

   GlobalDosFree (LOWORD(gdaBuffer));

   return (fResult);
}

/*---------------------------------------------------------------------
  MSCDEX_ReadSector()

  Calls MSCDEX to read a single sector from a CD-ROM compact disc.

  Parameters:

     bDrive
        Drive number of CD-ROM drive to read from. Expected to be
        a number in the following series: 0 = A, 1 = B, 2 = C, etc.

     dwStartSector
        First sector of read.

     RMlpBuffer
        Real-mode segment:offset pointer to a buffer that will receive
        sector data. Must be large enough to accomodate a single
        sector in cooked mode.

  Return Value
     Returns TRUE if successful, or FALSE if an error occurred in
     reading.
---------------------------------------------------------------------*/

BOOL FAR PASCAL MSCDEX_ReadSector (BYTE   bDrive,
                                   DWORD  dwStartSector,
                                   LPBYTE RMlpBuffer)
{
   RMCS   callStruct;
   BOOL   fResult;

   /*
      Prepare DPMI Simulate Real Mode Interrupt call structure with
      the register values used to make the MSCDEX Absolute read call.
      Then, call MSCDEX using DPMI and check for errors in both the DPMI
      call and the MSCDEX call.
   */
   BuildRMCS (&callStruct);
   callStruct.eax = 0x1508;                // MSCDEX Absolute read
   callStruct.ebx = LOWORD(RMlpBuffer);    // Offset of sect buffer
   callStruct.es  = HIWORD(RMlpBuffer);    // Segment of sect buffer
   callStruct.ecx = bDrive;                // 0=A, 1=B, 2=C, etc.
   callStruct.edx = 1;                     // Read one sector
   callStruct.esi = HIWORD(dwStartSector);
   callStruct.edi = LOWORD(dwStartSector);

   if (fResult = SimulateRM_Int (0x2F, &callStruct))
      fResult = !(callStruct.wFlags & CARRY_FLAG);
```

```
   return fResult;
}

/*---------------------------------------------------------------------
  SimulateRM_Int()

  Allows protected-mode software to execute real-mode interrupts
  such as calls to MS-DOS, MS-DOS TSRs, MS-DOS device drivers.

  This function implements the "Simulate Real Mode Interrupt"
  function of the DPMI specification v0.9 and later.

  Parameters:

     bIntNum
        Number of the interrupt to simulate.

     lpCallStruct
        Call structure that contains params (register values)
        for bIntNum.

  Return Value
     SimulateRM_Int returns TRUE if it succeeded or FALSE if
     it failed.

  Comments
     lpCallStruct is a protected-mode selector:offset address, not
     a real-mode segment:offset address.
---------------------------------------------------------------------*/

BOOL FAR PASCAL SimulateRM_Int (BYTE bIntNum, LPRMCS lpCallStruct)
{
   BOOL fRetVal = FALSE;        // Assume failure

   _asm {
         push di
         mov  ax, 0300h         ; DPMI Simulate Real Mode Interrupt
         mov  bl, bIntNum       ; Number of the interrupt to simulate
         mov  bh, 01h           ; Bit 0 = 1; all other bits must be 0
         xor  cx, cx            ; No words to copy from PM to RM stack
         les  di, lpCallStruct  ; Real mode call structure
         int  31h               ; Call DPMI
         jc   END1              ; CF set if error occurred

         mov  fRetVal, TRUE
     END1:
         pop di
        }
   return (fRetVal);
}

/*---------------------------------------------------------------------
   BuildRMCS()
```

```
   Initializes a real-mode call structure by zeroing all its members.

   Parameters:

      lpCallStruct
         Points to a real-mode call structure

   Return Value
         None.

   Comments
         lpCallStruct is a protected-mode selector:offset address,
         not a real-mode segment:offset address.
-------------------------------------------------------------------*/

void FAR PASCAL BuildRMCS (LPRMCS lpCallStruct)
   {
   _fmemset (lpCallStruct, 0, sizeof(RMCS));
   }
```

REFERENCES
==========

DOS Protected Mode Interface Specification Version 0.9.
Programmer's Guide to Windows 95.
Win32 SDK Online Documentation.

Additional reference words: Windows 95 4.00 cdrom cd-rom absolute
KBCategory: kbprg kbcode kbhowto
KBSubcategory: MMCDROM

How Win32-Based Applications Are Loaded Under Windows

PSS ID Number: Q110845

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

SUMMARY
=======

Microsoft Win32s is an operating system extension that allows Win32-based applications to be run on Windows 3.1. Win32s consists of a VxD and a set of dynamic-link libraries (DLLs).

It is possible to distinguish whether an executable was built for Win32 or Win16. Win32 executables use the Portable Executable (PE) format, while Win16 executables use the New Executable (NE) format.

The Windows 3.1 loader was designed to be aware that Win32-based applications would potentially be loaded. When the user starts a Win32-based application, the Windows 3.1 loader tries to load the Win32-based application via WinExec(). WinExec() calls LoadModule(), which will fail with an error indicating that it was passed an .EXE with the PE format. At this point, WinExec() calls a special function to start up Win32s. This function loads W32SYS.DLL (16-bit DLL) via LoadModule(). If W32SYS determines that the EXE is indeed a valid PE file, it calls LoadModule() on WIN32S.EXE (16-bit EXE) (it is similar to WinOldApp for MS-DOS-based programs running in Windows). WIN32S.EXE contains the task database, PSP, task queue, and module database. WIN32S.EXE calls its only function to load the Win32s 16-bit translator DLL, W32S16.DLL, which does work as a translator between the Win32-based application and the 16-bit world that it is running in.

MORE INFORMATION
================

Win32-based applications are loaded in the upper 2 gigabytes (GB) of the 4 GB address space under Win32s, whereas Windows NT loads them in the lower 2 GBs. This is because W32S.386, a VxD, allocates the memory, and VxDs get memory in the 2 GB to 4 GB range. The first 64K and the last 64K cannot be read or written to (similar to a null page on other architectures).

On Windows 3.1, all applications, even the Win32-based applications, share the same address space, unlike Windows NT where each Win32-based application gets its own address space. Each Windows-based application may be given its own address space, starting with Windows NT 3.5.

Additional reference words: 1.00 1.10 1.20 G byte
KBCategory: kbprg
KBSubcategory: W32s

How Win32-Based Applications Read CD-ROM Sectors in Windows NT

PSS ID Number: Q138434

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  Microsoft Windows NT versions 3.50 and 3.51

SUMMARY

Some Win32-based applications, such as multimedia applications and games, need to read sectors directly from compact discs in order to implement custom access and read caching that will optimize access for specific purposes. This article provides information and example code that demonstrates how Win32-based applications can read sectors from compact discs in cooked mode in Windows NT.

Windows 95 does not support this method for reading sectors from compact discs. For more information about how to read sectors from compact discs in Windows 95, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q137813
   TITLE   : How Win32 Applications Can Read CD-ROM Sectors in Windows 95

MORE INFORMATION

The Windows NT CD-ROM File System (CDFS) currently supports compact discs formatted with either the ISO 9660 or High Sierra file systems. CDFS does not support other file systems, so Win32-based applications cannot read sectors from discs formatted with file systems other than ISO 9660 or High Sierra.

Steps to Read Sectors from Compact Discs

Win32-based applications can read sectors in cooked mode from compact disks in the same way they read sectors from floppy and hard disks. Because the reads are in cooked mode, no error correcting codes are included in the sector data that the application reads. Follow these three basic steps:

1. Use CreateFile to open a CD-ROM drive with the syntax \\.\X: where X is the letter of the CD-ROM drive.

2. Use ReadFile or ReadFileEx to read sectors.

3. Use CloseHandle to close the CD-ROM drive.

As with floppy and hard disks, all sector reads must start on sector boundaries on the compact disc and must be an integral number of sectors long. Furthermore, the buffers used for the reads must be aligned on

addresses that fall on sector boundaries. For example, because a sector on a compact disc is normally 2048 bytes, the buffer that receives the sector data must be a multiple of 2048 and must start on an address that is a multiple of 2048. An easy way to guarantee the that the buffer will start on a multiple of 2048 is to allocate it with VirtualAlloc. Finally, although sectors on compact discs are normally 2048 bytes, you should use the DeviceIoControl IOCTL_CDROM_GET_DRIVE_GEOMETRY command to return the sector size to avoid hard-coded limits.

Issuing DeviceIoControl IOCTL Commmands to CD-ROM Drives

Whenf reading sectors from compact discs, applications usually need to use a few support functions that provide capabilities such as determining the characteristics of the media and locking the media in the drive so that it can't be removed accidentally. These functions are provided by DeviceIoControl IOCTL commands.

To issue IOCTL commands to CD-ROM drives, Win32-based applications must use the IOCTL compact disc commands defined in Ntddcdrm.h instead of the IOCTL disk commands defined in Winioctl.h. The IOCTL disk commands will fail if issued for compact discs. Documentation for the IOCTL compact disc commands is located in the Windows NT DDK. The Ntddcdrm.h header file is located in the Windows NT DDK in the \Ddk\Src\Storage\Inc directory.

Example Code that Reads Sectors from a Compact Disc

The following code demonstrates how to read sectors from a compact disc from a Win32-based application running on Windows NT.

```
#include <windows.h>
#include <winioctl.h>  // From the Win32 SDK \Mstools\Include
#include "ntddcdrm.h"  // From the Windows NT DDK \Ddk\Src\Storage\Inc

/*
   This code reads sectors 16 and 17 from a compact disc and writes
   the contents to a disk file named Sector.dat
*/

{
   HANDLE  hCD, hFile;
   DWORD   dwNotUsed;

   //  Disk file that will hold the CD-ROM sector data.
   hFile = CreateFile ("sector.dat",
                       GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
                       FILE_ATTRIBUTE_NORMAL, NULL);

   // For the purposes of this sample, drive F: is the CD-ROM
   // drive.
   hCD = CreateFile ("\\\\.\\F:", GENERIC_READ,
                     FILE_SHARE_READ|FILE_SHARE_WRITE,
                     NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
                     NULL);
```

```
   // If the CD-ROM drive was successfully opened, read sectors 16
   // and 17 from it and write their contents out to a disk file.
   if (hCD != INVALID_HANDLE_VALUE)
   {
      DISK_GEOMETRY         dgCDROM;
      PREVENT_MEDIA_REMOVAL pmrLockCDROM;

      // Lock the compact disc in the CD-ROM drive to prevent accidental
      // removal while reading from it.
      pmrLockCDROM.PreventMediaRemoval = TRUE;
      DeviceIoControl (hCD, IOCTL_CDROM_MEDIA_REMOVAL,
                       &pmrLockCDROM, sizeof(pmrLockCDROM), NULL,
                       0, &dwNotUsed, NULL);

      // Get sector size of compact disc
      if (DeviceIoControl (hCD, IOCTL_CDROM_GET_DRIVE_GEOMETRY,
                           NULL, 0, &dgCDROM, sizeof(dgCDROM),
                           &dwNotUsed, NULL))
      {
         LPBYTE lpSector;
         DWORD  dwSize = 2 * dgCDROM.BytesPerSector;  // 2 sectors

         // Allocate buffer to hold sectors from compact disc. Note that
         // the buffer will be allocated on a sector boundary because the
         // allocation granularity is larger than the size of a sector on a
         // compact disk.
         lpSector = VirtualAlloc (NULL, dwSize,
                                  MEM_COMMIT|MEM_RESERVE,
                                  PAGE_READWRITE);

         // Move to 16th sector for something interesting to read.
         SetFilePointer (hCD, dgCDROM.BytesPerSector * 16,
                         NULL, FILE_BEGIN);

         // Read sectors from the compact disc and write them to a file.
         if (ReadFile (hCD, lpSector, dwSize, &dwNotUsed, NULL))
            WriteFile (hFile, lpSector, dwSize, &dwNotUsed, NULL);

         VirtualFree (lpSector, 0, MEM_RELEASE);
      }

      // Unlock the disc in the CD-ROM drive.
      pmrLockCDROM.PreventMediaRemoval = FALSE;
      DeviceIoControl (hCD, IOCTL_CDROM_MEDIA_REMOVAL,
                       &pmrLockCDROM, sizeof(pmrLockCDROM), NULL,
                       0, &dwNotUsed, NULL);

      CloseHandle (hCD);
      CloseHandle (hFile);
   }
}
```

Additional reference words: 3.5 3.51 sector cdrom cd-rom
KBCategory: kbprg kbmm kbhowto kbcode

KBSubcategory: MMCDROM

Icons for Console Applications

PSS ID Number: Q91150

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Under OS/2, when adding an application named CONAPP.EXE to a program group, the system uses the file CONAPP.ICO (if it exists) as the icon. This does not happen automatically under Windows NT and Windows 95; the item will have a generic icon.

To specify the icon that appears in the program group, use the following steps:

1. Create a resource file containing an ICON statement:

      ConApp ICON ConApp.ICO

2. Compile the resource using RC:

      rc -r $(rcvars) -fo conapp.res conapp.rc

3. Add the .rc file to the link command line

MORE INFORMATION
================

If the application is started by clicking its icon in Program Manager, the icon that appears when the application is minimized will be that icon, whether it is a generic icon or an icon imbedded in the executable file.

If the application is started from the MS-DOS prompt or the File menu, then the icon that appears when the application is minimized will be the icon that is used for the MS-DOS prompt.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbprg
KBSubcategory: BseCon

Ideas to Remember as You Convert from ASCII or ANSI to Unicode

PSS ID Number: Q130052

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Win32s version 1.2

SUMMARY

Because the industry is moving to Windows NT, and Windows NT supports Unicode, many independent software vendors (ISVs) want to know how to convert existing ASCII (or ANSI) help and resource files into Unicode. This article gives you some notes on the subject.

MORE INFORMATION

When converting ASCII to Unicode, remember that the entire ASCII characters map perfectly to the first characters in Unicode. You need only add a second null bit and a 0x00 in the high byte.

When converting ANSI to Unicode, UCONVERT.EXE sources in the Win32 SDK and the good illustration of win32 MultiBytetoWideChar conversion API are helpful sources to consult.

When converting a Help file, you face the challenge presented by the fact that the old ANSI RTF format is clueless about wide characters. Also, you need an RTF format for each specific country. For example, you'll need one specific to Japan, another separate RTF format for Korea, and so on. In addition, you might want to consider converting to the newer, much more powerful Help file formats supported in the Windows 95 Help file system; this may be an easier solution.

Help files in Windows NT versions 3.1 and 3.5 use the same Help file format as Windows version 3.1 does. But newer operating systems such as Windows 95 contains a new help file compiler, engine, and format. One reason that Microsoft made this change was to improve international support.

One final idea you might want to consider is to develop your own converter by using the MutltiByteToWideChar() API. In fact, this may be the best approach, because application developers know exactly what kind of user interface they want to implement.

Additional reference words: 1.20 3.50 kbinf
KBCategory: kbother
KBSubcategory: wintldev

Identifying a Previous Instance of an Application

PSS ID Number: Q106385

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The entry point of both Windows and Windows NT applications is documented to be:

```
   int WinMain( hInstance, hPrevInstance, lpszCmdLine, nCmdShow )

   HINSTANCE hInstance;        /* Handle of current instance  */
   HINSTANCE hPrevInstance;    /* Handle of previous instance */
   LPSTR lpszCmdLine;          /* Address of command line     */
   int nCmdShow;               /* Show state of window            */
```

However, under Windows NT, hPrevInstance is documented to always be NULL. The reason is that each application runs in its own address space and may have the same ID as another application.

To determine whether another instance of the application is running, use a named mutex. If opening the mutex fails, then there are no other instances of the application running. FindWindow() can be used with the class and window name. However, note that a second instance of the application could be started, and could execute the FindWindow() call before the first instance has created its window. Use a named object to ensure that this does not happen.

MORE INFORMATION
================

The fact that hPrevInstance is set to NULL simplifies porting Win16 applications. Most 16-bit Windows-based applications contain the following logic:

```
   if( !hPrevInstance )
      if( !InitApplication(hInstance) )
         return FALSE;
```

Under Windows, window classes only are registered by the first instance of an application. Consequently, if hPrevInstance is not NULL, then the window classes have already been registered and InitApplication() is not called.

Under Windows NT, because hPrevInstance is always NULL, InitApplication() is always called, and each instance of an application will correctly register its window classes.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMisc

Identifying the Versions of International Windows

PSS ID Number: Q130062

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Win32s version 1.2

SUMMARY
=======

This article suggests three ways to check for the language used in an international version of a Windows-based application.

MORE INFORMATION
================

Option One: Least Coding and Least Accurate

Check the "sCountry" entry under the [intl] section of the WIN.INI file by using the GetProfileString API. It is likely that this will match the Windows language version. For example, German Windows will probably have "Deutchland" and English Windows will probably have "United States" or "United Kingdom." However, because the user can change this setting by using the Control Panel, it is not always accurate.

Option Two: Most Coding and Most Accurate

Check the "deflang" entry under the [data] section of the SETUP.INF file. This is a three-letter language code that SETUP.EXE uses. The setting will be one of these:

  English=ENU or ENG
  Spanish=ESP
  German=DEU
  French=FRA or FRC (French Canadian)
  Italian=ITA

The problem with this method is getting at the "deflang" entry in the SETUP.INF file. The applications should parse SETUP.INF. It's not that difficult, but it does involve extra coding.

Option Three: Let the User Choose

Suggest to the user what the application found, and let the user make final decision. Here's the algorithm:

   if Windows Version < 3.1

```
            Look at Win.ini, Setup.inf files
            Suggest a good guess and let the user choose;
            Register a profile string for your app;
   else
            Use version stamping;
```

For Windows version 3.1, the way to identify the character set is to use the version stamping API. The translation value from the GetFileVersionInfo when performed on GDI or SHELL.DLL is the only way in version 3.1 to find out the character set of the system. Please refer to the SDK documentation for more details on this API. Look for both GetFileVersionInfo and VERSIONINFO.

The VERSIONINFO statement creates a version-information resource. The resource contains information about the file such as its version number, its intended operating system, and its original filename. One of the parameters is langID, which specifies the language identifiers.

On Windows NT, look for the version resource information in one of the system DLLs, such as KERNEL32.DLL. Look at result returned by GetFileVersionInfo and its associated structures and references in the Win32 API help file or documentation. This should be correct on Windows NT version 3.5.

Additional reference words: 1.20 3.10 3.50 kbinf
KBCategory: kbother
KBSubcategory: wintldev

IME (Input Method Editor) Usage in Windows 95

PSS ID Number: Q118496

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Japanese Windows 95

SUMMARY

IMEs (Input Method Editors) are dynamic-link libraries that allow users to type complex ideographic characters using a standard keyboard. IMEs are available in Asian versions of the Microsoft Windows operating system, and help minimize the effort needed by users to enter text containing characters from Unicode and double-byte character sets (DBCS). IMEs relieve users of the need to remember all possible character values. Instead, IMEs monitor the user's keystrokes, anticipate the character the user may want, and present a list of candidate characters from which to choose.

MORE INFORMATION

Asian versions of Windows 3.1 and Windows NT each have a separate IME application programming interface (API). However, these APIs have been merged into a single API for Windows 95.

Applications for Far East versions of Windows 95 can choose from three levels of IME support:

1. IME-unaware: Merely retrieves double-byte characters through two WM_CHAR messages.

2. IME-aware: Takes control of the IME module's default user interface and properly handles Kanji strings passed to it by the IME.

3. Fully IME-aware: Controls the entire process of composing characters, including the display of intermediate keystrokes, and can customize the IME user interface.

IME, by default, provides an IME window through which users enter keystrokes and view and select candidate characters. Applications developed for the WIN32 APIs can use the Input Method Manager(IMM) functions and messages to create and manage their own IME windows, providing a custom interface while using the conversion capabilities of the IME.

Additional reference words: 4.00 international IME IMM
KBCategory: kbother
KBSubcategory: WIntlDev

IME Setup in Far East Windows 95

PSS ID Number: Q140897

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY
=======

On most Far East versions (Korean, Simplified Chinese, Traditional Chinese) of Windows 95, a user can add or delete additional IME (Input Method Editor) or language layouts by using the control panel keyboard language Add(D) or Remove(M)

MORE INFORMATION
================

IMEs for Windows 3.1 FE versions (all excecpt Japanese version) can also be added to Windows 95 by using control panel keyboard language Add 3.1 IME.

However, Japanese IMEs for the Windows 95 Japanese version cannot be manipulated this way. All Japanese IMEs must be installed or removed by its own setup program that comes with the IME. Language layouts can still be added or removed by using the control panel, keyboard icon, language page.

Additional reference words: front-end processor FE Asian kbinf
KBCategory: kbprg kbhowto
KBSubcategory: wintldev

Implementing a Line-Based Interface for Edit Controls

PSS ID Number: Q92626

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

In specific situations, it may be desirable to make multiline edit controls behave similar to list boxes, such that entries can be selected and manipulated on a per-line basis. This article describes how to implement the line-based interface.

MORE INFORMATION
================

A multiline edit control must be subclassed to achieve the desired behavior. The subclass function is outlined below.

Most of the work necessary to implement a line-based interface is done by the predefined window function of the edit control class. With the return value from the EM_LINEINDEX message, the offset of the line under the caret can be determined; with the length of that line retrieved via the EM_LINELENGTH message, the EM_SETSEL message can be used to highlight the current line.

There are two problems with this approach:

- The first problem is that the EM_LINEINDEX message, when sent to the control with wParam=-1, returns the line index of the caret, which is not necessarily the same as the current mouse position. Thus, upon receiving the WM_LBUTTONDOWN message, the subclass function should first call the old window function, which will set the caret to the character under the current mouse position, then compute the beginning and ending offsets of the corresponding line, and eventually set the selection to that line.

- The other problem is that the WM_MOUSEMOVE message should be ignored by the subclassing function because otherwise the built-in selection mechanism will change the selection when the mouse is being dragged with the left mouse button pressed, thus defeating the purpose.

Following is the subclassing function that follows from this discussion:

```
WNDPROC EditSubClassProc(HWND hWnd,
               UINT wMsg,
```

```
               WPARAM wParam,
               LPARAM lParam)
{ int iLineBeg, iLineEnd;
  long lSelection;
  switch (wMsg)
  { case WM_MOUSEMOVE:
     break;                        /* Swallow mouse move messages.  */
    case WM_LBUTTONDOWN:           /* First pass on, then process.  */
     CallWindowProc((FARPROC)lpfnOldEditFn,hWnd,wMsg,wParam,lParam);
     iLineBeg = SendMessage(hWnd,EM_LINEINDEX,-1,0);
     iLineEnd=iLineBeg+SendMessage(hWnd,EM_LINELENGTH,iLineBeg,0);
#ifndef WIN32
     SendMessage(hWnd,EM_SETSEL,0,MAKELPARAM(iLeneBeg,iLineEnd));
#else
     SendMessage(hWnd,EM_SETSEL,iLineBeg,iLine)  /* Win 32 rearranges
                     parameters.  */
#endif
     break;
    case WM_LBUTTONDBLCLK:
     lSelection = SendMessage(hWnd,EM_GETSEL,0,0);
     /* Now we have the indices to the beginning and end of the line in
        the LOWORD and HIWORD of lSelection, respectively.
        Do something with it... */
     break;
    default:

return(CallWindowProc((FARPROC)lpfnOldEditFn,hWnd,wMsg,wParam,lParam));
};
return(0);
}
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Importance of Calling DefHookProc()

PSS ID Number: Q74547

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

When an application installs a hook using SetWindowsHook(), Windows adds the hook's callback filter function to the hook chain. It is the responsibility of each callback function to call the next function in the chain. DefHookProc() is used to call the next function in the hook chain for Windows 3.0. DefHookProc() is retained in Windows 3.1 for backwards compatibility. For Windows 3.1, you should use CallNextHookEx() to call the next function in the hook chain.

For Win32, mouse and keyboard hooks can suppress messages by return value and do not have to call CallNextHookEx(), unless they want to pass the message on. Other hooks, like WH_CALLWNDPROC, don't need to call CallNextHookEx(), because it will be called by the system. However, all hooks should call CallNextHookEx() immediately if nCode<0.

MORE INFORMATION
================

Windows 3.0

If a callback function does not call DefHookProc(), none of the filter functions that were installed before the current filter will be called. Windows will try to process the message and this could hang the system.

Only a keyboard hook (WH_KEYBOARD) can suppress a keyboard event by not calling DefHookProc() and returning a 1. When the system gets a value of 1 from a keyboard hook callback function, it discards the message.

Windows 3.1

In Windows 3.1, the WH_MOUSE hook will work like the WH_KEYBOARD hook in that the mouse event can be suppressed by returning 1 instead of calling DefHookProc().

Furthermore, when the hook callback function receives a negative value for the nCode parameter, it should pass the message and the parameters to DefHookProc() without further processing. When nCode is negative, Windows

is in the process of removing a hook callback function from the hook chain.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrHks

Improve System Performance by Using Proper Working Set Size

PSS ID Number: Q126767

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.5

SUMMARY
=======

While increasing your working set size and locking pages in physical memory can reduce paging for your application, it can adversely affect the system performance. When making decisions for your application, it is important to consider the whole system, and then test your application under a heavily loaded system, such as the users of your application might have.

MORE INFORMATION
================

The Win32 SDK provides a tool called the working set tuner (WST.EXE). The working set tuner decreases the working set size, which decreases memory demand. However, you can also choose to set the process working set minimum and maximum using SetProcessWorkingSetSize() and/or lock pages into memory with VirtualLock(). These APIs should be used with care. Suppose you have a 16-megabyte system and you set your minimum to four megabytes. In effect, this takes away four megabytes from the system. Other applications may be unable to get their minimum working set. You or other applications may be unable to create processes or threads or perform other operations that require non-paged pool. This can have an extremely negative impact on the overall system.

Reducing memory consumption is always a beneficial goal. If you call SetProcessWorkingSetSize(0xffffffff, 0xffffffff), this tells the system that your working set can be released. This does not change the current sizing of the working set, it just allows the memory to be used by other applications. It is a good idea to do this when your application goes into a wait state. When you call SetProcessWorkingSet(0, 0), your working set is reset to the default values. In addition, if you call VirtualUnlock() on a range that was not locked, it is used as a hint that those pages can be removed from the working set.

Additional reference words: 3.50
KBCategory: kprg
KBSubcategory: BseMm

Improving the Performance of MCI Wave Playback

PSS ID Number: Q77700

The information in this article applies to:

- Microsoft Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

This article discusses two methods to improve playback performance for a series of MCI wave files in an application developed for the Microsoft Windows environment.

MORE INFORMATION

The following code fragment demonstrates opening the device and wave file at the same time. This method does not give the best performance.

```
   mciopen.lpstrDeviceType = (LPSTR)"waveaudio";
   mciopen.lpstrElementName = (LPSTR)lpWavefile;

   // The following two fields must be initialized or the debugging
   // version of MMSYSTEM will cause an unrecoverable application
   // error (UAE).
   mciopen.lpstrDeviceType = "\0";
   mciopen.lpstrAlias = "\0";

   dwFlags = MCI_OPEN_TYPE | MCI_OPEN_ELEMENT;

   dwRes = mciSendCommand(0, MCI_OPEN, dwFlags,
                          (DWORD)(LPSTR)&mciopen);
```

To improve performance, open the device separately from the wave file (element) and leave the device open until the last element in the series has been played. Alternately, open and close elements but leave the global (waveaudio) device open during the entire process. The following code fragment demonstrates this process:

```
   // Open the waveaudio driver separate from the element.
   mciopen.lpstrDeviceType = (LPSTR)MCI_DEVTYPE_WAVEFORM_AUDIO;
   dwFlags = MCI_OPEN_TYPE;
   dwRes = mciSendCommand(0, MCI_OPEN, dwFlags,
                          (DWORD)(LPSTR)&mciopen);
```

The following code fragment demonstrates using the global device ID to open the wave file separately:

```
   dwFlags = MCI_OPEN_ELEMENT;
   mciopen.lpstrElementName = (LPSTR)lpWaveName;
   dwRes = mciSendCommand(wGlobalDeviceID, MCI_OPEN, dwFlags,
```

```
(DWORD)(LPSTR)&mciopen);
```

This allows the application to open and play wave files without incurring the performance penalty involved with opening the device. Another method to speed loading a wave file is to use the fully qualified path. For example, rather than specifying LASER.WAV, specify C:\MMWIN\MMDATA\LASER.WAV. If this is done, MCI is not required to search the directories in the MS-DOS PATH environment variable for the wave file.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbmm kbprg
KBSubcategory: MMWave

Increased Performance Using FILE_FLAG_SEQUENTIAL_SCAN

PSS ID Number: Q98756

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

There is a flag for CreateFile() called FILE_FLAG_SEQUENTIAL_SCAN which will direct the Cache Manager to access the file sequentially.

Anyone reading potentially large files with sequential access can specify this flag for increased performance. This flag is useful if you are reading files that are "mostly" sequential, but you occasionally skip over small ranges of bytes.

MORE INFORMATION

The effect on the Cache Manager of this flag is two-fold:

- There is a minor savings because the Cache Manager dispenses with keeping a history of reads on the file, and tries to maintain a high-water mark on read ahead, which is always a certain delta from the most recent read.

- More importantly, the Cache Manager reads further ahead for sequential access files--currently about three times more than files that are currently detected for sequential access.

If the caller makes multiple passes through a file, there are no negative effects of specifying the sequential flag, because the Cache Manager will still disable read ahead for as long as the application is getting hits on the file (such as on the second or subsequent pass).

If you are working on an application where your ability to sequentially read file data is key to performance, you may want to consider adding the sequential flag to your create file call. This is especially true of applications that use this flag to read from a CD-ROM.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseFileio

Initializing Menus Dynamically

PSS ID Number: Q75630

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Many commercial applications developed for Windows allow the user to customize the menus of the application. This ability introduces some complexity when the application must disable particular menu items at certain times. This article provides a method to perform this task.

Windows sends a WM_INITMENUPOPUP message just before a pop-up menu is displayed. The parameters to this message provide the handle to the menu and the index of the pop-up menu on the main menu.

To process this message properly, each menu item must have a unique identifier. When the application starts up, it creates a mapping array that lists the items on each menu. When the WM_INITMENUPOPUP message is received, the application checks the conditions necessary for each menu item to be disabled or checked and modifies the menu appropriately.

The application must maintain the mapping array when the user modifies the menus in any way.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMen

Installing the Win32s NLS Files

PSS ID Number: Q124136

The information in this article applies to:

- Microsoft Win32s version 1.2

SUMMARY

=======

The help file for Win32s version 1.2 states that the following NLS files need to be installed in the <windows>\SYSTEM\WIN32S directory when you install Win32s version 1.2 with your application. However, if your Win32-based application is only targeting U.S. Windows, you do not need to install all these files.

.NLS files:

CTYPE.NLS	P_850.NLS
LOCALE.NLS	P_852.NLS
L_INTL.NLS	P_855.NLS
P_037.NLS	P_857.NLS
P_10000.NLS	P_860.NLS
P_10001.NLS	P_861.NLS
P_10006.NLS	P_863.NLS
P_10007.NLS	P_865.NLS
P_10029.NLS	P_866.NLS
P_10081.NLS	P_869.NLS
P_1026.NLS	P_875.NLS
P_1050.NLS	P_932.NLS
P_1051.NLS	P_936.NLS
P_1252.NLS	P_949.NLS
P_1053.NLS	P_950.NLS
P_1054.NLS	SORTKEY.NLS
P_437.NLS	SORTTBLS.NLS
P_500.NLS	UNICODE.NLS
P_737.NLS	

MORE INFORMATION

================

Use the following guidelines when shipping a Win32-based application that targets Win32s:

Ship National Language Support (.NLS) files corresponding to the market of the Win32-based application. For a Win32-based application released for an international market, ship all the .NLS files found in MSTOOLS\WIN32S\NLS. For a Win32-based application released for use only in the United States, ship P_437.NLS, P_850.NLS, and P_1252.NLS.

NOTE: When installing Win32s, be sure to remove the obsolete files. These are tagged in WIN32S\SETUP\W32S.LYT with Win32sSystemObsoleteFiles.

Additional reference words: 1.20
KBCategory: kbusage kbpolicy kbdocerr
KBSubcategory: W32s

Instance-Specific String Handles (HSZs) in DDEML

PSS ID Number: Q94953

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Instance-specific string handles in DDEML may be used in DdeConnect() or DdeConnectList() in order to connect to a particular instance of a server. These string handles are received by a DDEML callback as the HSZ2 parameter to the XTYP_REGISTER/XTYP_UNREGISTER transactions whenever a server application registers or unregisters the service name it supports.

This article explains how instance-specific HSZs are internally implemented in the Windows 3.1 DDEML; however, this is for purposes of illustration only because the implementation may change in future versions of DDEML, particularly in Win32. However, the behavior will be the same.

MORE INFORMATION

Currently, instance-specific string handles contain two pieces of information: the original service name string plus the handle to a hidden window created by DDEML, which is associated with that string. These two pieces of information are then merged [that is, MAKELONG (SvcNameAtom, hWnd)] into an HSZ.

It is important to underscore what the documents on DdeCreateStringHandle() say in reference to instance-specific HSZs (see the Comments section of the Windows 3.1 Software Development Kit (SDK) "Programmer's Reference, Volume 2: Functions," page 169):

> An instance-specific string handle is not mappable from string handle to string to string handle again. The DdeQueryString() function creates a string from a string handle and then DdeCreateStringHandle() creates a string handle from that string, but the two handles are not the same.

This might be better explained as follows:

1. Server registers itself:

```
0x0000C18F = DdeCreateStringHandle (,"SERVER",);
DdeNameService (,0x0000C18F,,);
```

2. Callbacks receive two HSZs in XTYP_REGISTER:

```
HSZ1 = 0x0000C18F  (normal HSZ)
HSZ2 = 0x56F8C18F  (instance-specific HSZ)
```

3. Client does a DdeQueryString() on the HSZ2 returned above, and creates a string handle with the string returned.

```
DdeQueryString (,0x56F8C18F, myLpstr,,,);
      // where myLpstr returned = "SERVER:(56F8)"

0x0000C193= DdeCreateStringHandle (,myLpstr,);
```

Note how instance-specific 0x56F8C18F passed in to DdeQueryString() is not the same as the HSZ returned (0x0000C193) from the DdeCreateStringHandle() on the same string; whereas regular string handles (that is, non-instance-specific HSZs) would have mapped to the same string handle.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Intergraph's NFS causes WinSock APIs to return error 10093

PSS ID Number: Q127015

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SYMPTOMS
========

The presence of Intergraph's NFS (Network File System) may cause Windows Sockets applications to function incorrectly. Specifically, if the 'Choose File' common dialog is displayed in a Windows Sockets application, subsequent Windows Sockets API calls may return error 10093, WSAENOTINITIALIZED.

STATUS
======

This problem has been corrected in an updated version of PC-NFS, version 2.0.8.0. The update is available from Intergraph's FTP server or bulletin board (IBBS): (205) 730-7248. For more information, please call Intergraph technical support at (800) 633-7248.

MORE INFORMATION
================

Steps to Reproduce Problem
---

1. Call WSAStartup(...).

2. Call any WinSock API function to verify that WSOCK32.DLL is initialized.

3. Call GetOpenFileName to display the 'Open File' common dialog.

4. Call any WinSock API function. If Intergraph's NFS implementation is installed, you may receive error 10093.

Additional reference words: 3.50 Windows Sockets
bug CFileDialog
KBCategory: kbnetwork
KBSubcategory: NtwkWinsock

International Glossaries Available on MSDN

PSS ID Number: Q140764

The information in this article applies to:

- Microsoft Win32 Software Development Kit for
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

To promote the use of common terminology in the international developer and localization communities, Microsoft is making its international language glossaries available to the software development community. Right now, the glossaries are only available on Microsoft Developer Network, level II subscription (Development Platform) compact discs.

MORE INFORMATION

On compact disc No. 13 "INTL: Additional Windows NT Service Packs and Windows 3.11 Versions," under \Glossary directory, you will find the following language versions of glossaries:

Brazilian-Portuguese
Czech
Danish
Dutch
Finnish
French
German
Greek
Hungary
Italian
Japanese
Norwegian
Polish
Portuguese
Simplified Chinese
Russian
Slovak
Spanish
Swedish
Traditional Chinese
Turkish

The glossary collection will be updated with each release of the MSDN compact discs. Additional glossaries will be added when they become available.

Additional reference words: 4.00 translation user interface menu string

globalization
KBCategory: kbref
KBSubcategory: wintldev

International Versions of Windows 95

PSS ID Number: Q118495

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

SUMMARY

=======

Microsoft Windows version 3.1 is available in 27 different language versions. Windows 95 will also be released in 27 different language versions, with possibly more versions following later.

MORE INFORMATION

================

There are three categories of the Windows 95 platform:

1. Western and Eastern European languages, plus Indonesian, which are based on single-byte character sets (SBCSs) and are written from left to right:

   English
   German
   French
   Spanish
   Swedish
   Dutch
   Italian
   Norwegian
   Danish
   Finnish
   Portuguese-Brazil
   Portuguese-Portugal
   Russian
   Czech
   Polish
   Hungarian
   Turkish
   Greek
   Basque
   Catalan
   Indonesian

2. Middle East languages, plus Thai, which are based on SBCSs from both right to left and left to right:

   Arabic
   Hebrew
   Thai

3. Far East languages, which are based on double-byte character

sets (DBCSs) and are written from both left to right and, in certain types of applications, from top to bottom:

- Japanese
- Korean
- Simplified Chinese
- Traditional Chinese

Additional reference words: 4.00 versions international
KBCategory: kbother
KBSubcategory: WIntlDev

Interpreting Executable Base Addresses

PSS ID Number: Q101187

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

LINK.EXE and DUMPBIN.EXE (from Visual C++ 32-bit edition) can be used to dump the portable executable (PE) header of an executable file. Below is a fragment of a dump:

```
   7300 address of entry point
   7000 base of code
   B000 base of data
        ----- new -----
   10000 image base
```

The "image base" value of 10000 is the address where the program begins in memory. The value associated with "base of code," "base of data," and "address of entry point" are all offsets from the image base.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsMisc

Interprocess Communication on Windows NT, Windows 95, & Win32s

PSS ID Number: Q95900

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The following are some of the standard mechanisms available for interprocess communication (IPC): NetBIOS, mailslots, windows sockets (winsock), named pipes, anonymous pipes, semaphores, shared memory, and shared files. Other IPC mechanisms available on Microsoft systems include DDE, OLE, memory-mapped files, Windows messages, Windows atoms, the registration database, and the clipboard.

MORE INFORMATION

The table below denotes what platforms and subsystems provide which IPC mechanisms (this does not imply that all the mechanisms will interoperate between different subsystems):

Interprocess Communication Mechanisms

IPC Mechanism	WinNT	Win95	Win32s(1)	Win16(2)	MS-DOS(2)	POSIX	OS/2
DDE	YES	YES	YES	YES	NO	NO	NO
OLE 1.0	YES	YES	YES	YES	NO	NO	NO
OLE 2.0	YES	YES	YES	YES	NO	NO	NO
NetBIOS	YES	YES	YES	YES	YES	NO	YES
Named pipes	YES	YES(3)	YES(3)	YES(3)	YES(3)	YES(4)	YES
Windows sockets	YES(5)	YES	YES	YES(5)	NO	NO(6)	NO
Mailslots	YES	YES	YES(3)	NO	NO	NO	YES
Semaphores	YES	YES	NO	NO	NO	YES	YES
RPC	YES	YES(7)	YES(8)	YES	YES	NO	NO
Mem-Mapped File	YES	YES	YES	NO	NO	NO	NO
WM_COPYDATA	YES	YES	YES(9)	YES	NO	NO	NO

(1) Win32s an extension to Windows 3.1, which allows Win32-based applications to run under Windows 3.1. Win32s supports all the Win32 APIs, but only a subset provides functionality under Windows 3.1. Those APIs that are not functional return ERROR_CALL_NOT_IMPLEMENTED.

(2) This is technically not a subsystem.

(3) Cannot be created on Win16, Windows 95 and MS-DOS workstations, but can

be opened.

(4) The POSIX subsystem supports FIFO queues, which do not interoperate with Microsoft's implementation of named pipes.

(5) Via the Windows sockets API.

(6) Currently BSD-style sockets are under consideration for the POSIX subsystem.

(7) Windows 95 supports the RPC 1 protocol only. The NetBios protocol is not supported. Namedpipe servers are not supported.

(8) Win32s version 1.1 provides network support through Universal Thunks.

(9) Under Win32s, WM_COPYDATA does not actually copy the data -- it only translates the pointers to the data. If the receiving application changes the buffer, then the data is changed for both applications.

(10) OLE objects created in a Win32 service must be in the same user context as a logged on user that wishes to use them. Any attempt to access these objects from a different user context will result in failure. For example, a service that runs under the LocalSystem account creates an object that an application running in Domain\User's context attempts to access will fail.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbprg
KBSubcategory: BseIpc

Interrupting Threads in Critical Sections

PSS ID Number: Q101193

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

If a thread enters a critical section and then terminates abnormally, the critical section object will not be released. Many components of the C Run-time library are not reentrant and use a resource locking scheme to maintain coherency in the multithreaded environment. Thus, a thread that has entered a C Run-time function, such as printf(), could deadlock all access (within that process) to printf() if it terminates abnormally.

This situation could arise if a thread is terminated with TerminateThread() while it holds a resource lock. If this occurs, any thread that tries to acquire that resource lock will become deadlocked.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

Inverting Color Inverts Palette Index, Not RGB Value

PSS ID Number: Q71227

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Performing any bitwise logical operator on a color, such as inversion, does not modify the color's RGB value; it instead changes the index into the system palette. This applies also to the ROP codes associated with the blt functions (BitBlt, StretchBlt, and PatBlt) and in the SetRop2 function. For display devices with hardware palettes [generally, devices with fewer than 24 bits-per-pixel (BPP)], this can produce unexpected or undesirable results.

MORE INFORMATION

Suppose the system palette contained the following colors:

	Entry #							
Color	0	1	2	3	4	5	6	7
Red =	0	0x80	0	0	0x80	0x80	0	0x80
Green =	0	0	0x80	0	0x80	0	0x80	0x80
Blue =	0	0	0	0x80	0	0x80	0x80	0x80

	Entry #							
Color	8	9	A	B	C	D	E	F
Red =	0xC0	0xFF	0	0	0xFF	0xFF	0	0xFF
Green =	0xC0	0	0xFF	0	0xFF	0	0xFF	0xFF
Blue =	0xC0	0	0	0xFF	0	0xFF	0xFF	0xFF

The inversion of colors would look like this:
(Half = half intensity, Full = full intensity)

Color	Index	Inverse Color	Index
Black	0	White	F
Half Red	1	Full Cyan	E
Half Green	2	Full Magenta	D
Half Blue	3	Full Yellow	C
Half Yellow	4	Full Blue	B

Half Magenta	5	Full Green	A
Half Cyan	6	Full Red	9
Dark Gray	7	Light Gray	8

This obviously is not the desired effect. Inverting a full-intensity color such as red will not invert to full-intensity cyan; instead, it is inverted to half-intensity cyan.

This is also true for any logical operations performed on the bits of a bitmap, pen, or flood fill through ROP codes. All operations are done on the index of the color and not its RGB value.

Note that when using custom palettes on a palette capable device, the application does not have control over the precise mapping between logical palette indexes and system palette indexes. The results of bitwise logical operations are unpredictable in such a case.

The only way for an application to precisely control color mixing is to perform the operation on RGB values, then translate the RGB result back into the most appropriate palette index.

For example, one way to do this is to mix colors in a 24 BPP device-independent bitmap (DIB), then set the DIB bits into the device context (DC) again when finished. Another method is to query the RGB color of pixels to modify, do the mixing, and then use the SetPixel function to apply the change to the DC.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiPal

Joliet Specification for CD-ROM

PSS ID Number: Q125630

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

Content authors who are developing Windows 95 applications on CD-ROM should develop their titles according to the Joliet specification in order to incorporate Unicode file names and take full advantage of Windows 95 long file name support.

The Joliet specification complies with the ISO 9660:1988 standard. It is designed to resolve some of its restrictions and ambiguities including:

- Character Set limitations.
- File Name Length limitations.
- Directory Tree Depth limitations.
- Directory Name Format limitations.
- Unicode Character ambiguities.

Because the Joliet specification is ISO 9660 compliant, CD-ROM disks recorded according to the Joliet specification may continue to interchange data with non-Joliet systems. The designs for the System Use Sharing Protocol, RockRidge extensions for POSIX semantics, CD-XA System Use Area Semantics, and Apple's Finder Flags and Resource Forks all port in a straightforward manner to the Joliet specification. These protocols are not an integral part of the Joliet specification, however.

Support for Joliet is included in Windows 95, and the spec is now included in the Win95 DDK docs. It will also be included in a future version of Windows NT. The Joliet spec is now available on the web at http://www.ms4music.com/devl/dvjoliet.htm (that's "devl" as in developer). The spec can also be downloaded from our FTP site: developr/drg/multimedia/joliet/joliet-ZIP.zip.

Additional reference words: 4.00 CD CD-ROM XA SUSP ROCKRIDGE LFN
KBCategory: kbprg
KBSubcategory: SubSys

Journal Hooks and Compatibility

PSS ID Number: Q106717

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

Journal hooks are used to record and play back events, such as keyboard and mouse events. Journal hooks are system-wide hooks that take control of all user input, and therefore should be used as little as possible.

MORE INFORMATION

================

Note that Windows NT does not ship with a Recorder application, as Windows 3.1 does. Therefore, it may be desirable to create an application that can play back macros recorded under Windows 3.1. However, there are a number of different problems with the Windows NT implementation of journaling that make it difficult to use macros recorded under Windows 3.1.

The EVENTMSG structures recorded under Windows 3.1 that hold keystrokes do not play back under Windows NT. They must be modified, because the journal playback hook parses a scan code out of the EVENTMSG structure differently than the Windows 3.1 journal record hook put it in the structure. Under Windows 3.1, paramH specifies the repeat count. Under Windows NT, there is no way to specify a repeat count; it is always assumed to be 1.

For more information on hooks, please see the Hooks Overview in Volume 1 of the Win32 "Programmer's Reference" and the article "Win32 Hooks" included in the MSDN CD #5.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrHks

Jumping to a Keyword in the Middle of a Help Topic

PSS ID Number: Q94611

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY
=======

The following information was extracted from the Windows Help Authoring Guide available on the Microsoft Developer's Network CD-ROM:

Placing Keywords in the Topic

When the user goes to a topic by choosing a keyword from the Search dialog box, Help displays the selected topic in the main window, starting from the beginning of the topic. If the information related to the keyword is located in the middle or toward the end of the topic, the user may not be able to see the relevant information without scrolling the topic. This result may not be what you want.

If you want users to be able to go directly to relevant information within a topic (and see it without scrolling), you can place additional keywords with the information you want users to find. Keywords placed within a topic function as spot references (similar to context-string spot references) because they index specific locations, or "spots," within the topic. To access the spot-referenced material, users choose the keyword from the Search dialog box.

In the Search dialog box, all keywords appear the same. The user cannot tell the difference between keywords placed at the beginning of the topic and those placed elsewhere in the topic. However, when the user goes to the topic, Windows Help uses the location of the keyword footnote as a reference point. If the keyword footnote is located in the middle of the topic, Help displays the topic as if the middle location were the "top" of the topic.

NOTE:

Because you cannot insert a title footnote in the middle of a topic, any keywords that you place in the middle of the topic use the main topic title in the Search dialog box.

To define a keyword in the middle of the topic:

1. Place the insertion point where you want to define the keyword. A keyword inserted anywhere except the beginning of the topic is treated as a spot reference.

2. From the Insert menu, choose Footnote. The Footnote dialog box appears.

3. Type an uppercase K as the custom footnote mark, and then choose OK. A superscript K appears in the text window, and the insertion point moves to the footnote window.

4. Type the keyword(s) to the right of the K in the footnote window. Use only a single space between the K and the first keyword. Separate multiple keywords with a semicolon (;).

Additional reference words: 3.10 3.50 4.00 95 winhelp hc
KBCategory: kbprg
KBSubcategory: TlsHlp

LANGUAGE Statement in RC Differs in Windows NT from Windows 95

PSS ID Number: Q140763

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3
- Microsoft Win32 Software Development Kit (SDK) for:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The LANGUAGE statement in resource files sets the language for all resources up to the next LANGUAGE statement or to the end of the file. When the LANGUAGE statement appears before the BEGIN in an ACCELERATORS, DIALOG, MENU, RCDATA, or STRINGTABLE resource definition, the specified language applies only to that resource.

The general syntax for the language statement is:

```
   LANGUAGE language, sublanguage
```

MORE INFORMATION

In Windows NT, several language versions of the same resource (with the same ID and type) can be loaded at the same time. The programmer can decide which language version of the resource to use dynamicaly by calling SetThreadLocale(). For example, the following is in a RC file:

```
ABOUTBOX DIALOG DISCARDABLE  52, 57, 144, 45
LANGUAGE LANG_GERMAN, SUBLANG_GERMAN
STYLE DS_MODALFRAME | WS_CAPTION
FONT 8, "System"
BEGIN
    CTEXT           "Microsoft(r)",-1,0,5,144,8
    CTEXT           "German menu Example",-1,0,14,144,8
    ICON            IDR_MENU,-1,10,10,18,20
    DEFPUSHBUTTON   "OK",IDOK,53,25,32,14,WS_GROUP
END

ABOUTBOX DIALOG DISCARDABLE  52, 57, 144, 45
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
STYLE DS_MODALFRAME | WS_CAPTION
FONT 8, "System"
BEGIN
```

```
    CTEXT           "Microsoft(r)",-1,0,5,144,8
    CTEXT           "Menu Example",-1,0,14,144,8
    ICON            IDR_MENU,-1,10,10,18,20
    DEFPUSHBUTTON   "OK",IDOK,53,25,32,14,WS_GROUP
END
```

In Winodws NT, you can call SetThreadLocale (MAKELCID(MAKELANGID(LANG_GERMAN, SUBLANG_GERMAN), SORT_DEFAULT)) to set the thread locale to German, and the German resource (the first dialog box) would is automatically used.

In Windows 95, multiple copies of a resource with exactly the same ID and resource type cannot be loaded at the same time. Thus, the only value of LANGUAGE that works is the one that matches the current locale. In the previous example, if the current system locale is German, then only the German dialog box will be loaded.

Additional reference words: 4.00 1.30 setlocale kbinf
KBCategory: kbprg
KBSubcategory: wintldev

LB_GETCARETINDEX Returns 0 for Zero Entries in List Box

PSS ID Number: Q97922

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

To determine whether a multiple selection list box is empty or has no items to select, two messages are required. First, call LB_GETCOUNT to determine whether or not the list box is empty. If the list box is not empty, then use LB_GETCARETINDEX to determine the position of the caret.

If you want a list box to contain selections that remain after the focus goes elsewhere, Microsoft recommends using visible check marks next to the items in the list box. This method provides better visual feedback to the user than a selection bar.

Additional reference words: 3.50 3.51 4.00 95 listbox
KBCategory: kbui
KBSubcategory: UsrCtl

Length of STRINGTABLE Resources

PSS ID Number: Q20011

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

In order to find the length of a string in the STRINGTABLE you need to do a FindResource() and then a SizeofResource() to find the total size in bytes of the current block of 16 strings. Remember that STRINGTABLEs are stored specially; to FindResource() you will ask for RT_STRING as the Type, and the (string number / 16) + 1 as the name.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrRsc

Limitations of Overlapped I/O in Windows 95

PSS ID Number: Q125717

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Windows 95 does not support overlapped operations on files, disk, pipes, or mailslots, but does support overlapped operations on serial and parallel communication ports. Non-overlapped file write operations are similar to overlapped file writes, because Windows 95 uses a lazy-write disk cache. Overlapped I/O can be implemented in a Win32-based application by creating multiple threads to handle I/O.

MORE INFORMATION
================

Asynchronous I/O on files, disk, pipes and mailslots is not implemented in Windows 95. If a Win32-based application running on Windows 95 attempts to perform asynchronous file I/O (such as ReadFile() with any value other than NULL in the lpOverlapped field) on any of these objects, the ReadFile() or WriteFile fails and GetLastError() returns ERROR_INVALID_PARAMETER (87).

Overlapped I/O to serial and parallel communication ports is fully supported in Windows 95. To implement overlapped I/O, call CreateFile() with the FILE_FLAG_OVERLAPPED flag set in the flags attribute.

Overlapped I/O on disk and files was not implemented in Windows 95, because the added performance benefits (which would only affect a certain class of I/O-intensive applications) were not judged to be worth the extra cost.

The system uses a lazy-write disk cache algorithm which automatically provides many of the benefits of overlapped writes. When a process writes data to a file, the data is written to the cache and then the write immediately returns to the calling process. Then, at some later time, the cache manager writes the data to disk. This is similar to behavior that is achieved with overlapped I/O. In the case of disk/file reads, many applications that need to read data do so because it is needed for further processing. Some applications benefit greatly by prefetching data from files while doing other work.

Although Windows 95 does not implement overlapped I/O, it is possible for Win32-based applications on Windows 95 to create additional threads for implementing an effect similar to overlapped file I/O. One way to implement this effect is to communicate with an I/O thread using a request packet mechanism. The thread can queue request packets from other threads and service them as it is able, signalling the other threads on the completion of each request with an event. Even though the application may be "waiting"

on some I/O activity, it can still be responsive to the user since the main, or user interface, thread is not blocking on I/O requests. However, the use of multiple threads will increase the amount of time spent in the kernel, which leads to inefficiencies, as compared with an operating system that supports overlapped I/O.

NOTE: Windows NT has a lazy-write disk cache as well. It has been found that a write-back cache is not as good as overlapped I/O, particularly when the data is much larger than the file cache or noncached I/O. One of the biggest benefits of overlapped I/O is that it allows you to quickly get lots of outstanding I/O posted to the disk controller, thereby keeping the disks busy with tagged command queueing in the controller. Using multiple threads is a reasonable substitute, but I/O may be serialized in the filesystem.

Additional reference words: 4.00 95 asynchronous overlapped
KBCategory: kbprg
KBSubcategory: BseFileio

Limitations of POSIX Applications on Windows NT

PSS ID Number: Q149902

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), for Windows NT, version 3.5, 3.51, 4.0

SUMMARY
=======

This article discusses the limitations of the Portable Operating System Interface (POSIX) applications on Windows NT. POSIX is a standard set by ANSI/IEEE to promote source level compatibility that allows applications to run on a wide variety of systems and architectures. The POSIX interface on Windows NT strictly follows the POSIX 1003.1-1990 standards.

MORE INFORMATION
================

Following are some of the POSIX limitations:

- POSIX applications only launch other POSIX applications. They do not launch DOS, OS/2, Win16 or Win32 applications.

- POSIX applications do not call any APIs. They do not access DDE, OLE, memory mapped files, named pipes, windows sockets and other Win32 features.

- POSIX applications do not implicitly or explicitly load a Win32 DLL.

- POSIX applications do not have access to any networking APIs such as pipes or sockets. They are not network aware, but they do access files over the network.

- POSIX applications do not have any source level debugger support. You cannot use Windbg or the Microsoft Visual C++ debugger to debug POSIX applications on Windows NT.

For additional information, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q99361
TITLE : Specifying Filenames Under the POSIX Subsystem

REFERENCES
==========

MSDN Development Library, "Understanding Windows NT POSIX Compatibility", by Ray Cort.

Additional reference words: 3.50 3.51 4.00 POSIX
KBCategory: kbprg

KBSubcategory: SubSys

Limiting the Number of Entries in a List Box

PSS ID Number: Q78241

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Although there is no single message that restricts the number of entries (lines) allowed in a list box, the limit can be imposed through the use of subclassing.

MORE INFORMATION
================

The following code fragment is an excerpt from a subclassing function that can be used to restrict the number of entries in a list box to no more than the constant MAXENTRIES where the lpfnOldLBFn variable points to the original list box window procedure:

```
long FAR PASCAL SubClassFn(hWnd, message, wParam, lParam)
HWND hWnd;
unsigned message;
WORD wParam;
LONG lParam;
{
   int iCount;

   switch (message)
      {
   case LB_ADDSTRING:
   case LB_INSERTSTRING:
      iCount = SendMessage(hWnd, LB_GETCOUNT, 0, 0L);
      if (iCount > MAXENTRIES)
         { /* Insert action here to inform user of limit violation */
         break;
         }
      /* fall through if less entries than maximum */

   default:
      return CallWindowProc(lpfnOldLBProc, hWnd, message, wParam,
            lParam);
      }
}
```

Additional reference words: 3.00 3.10 3.50 4.00 95 list box

KBCategory: kbui
KBSubcategory: UsrCtl

Limits on Overlapped Pipe Operations

PSS ID Number: Q115522

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

The Windows NT version 3.1 redirector allows only 17 outstanding overlapped pipe operations at any given time. The Windows NT 3.5 redirector does not have this restriction on overlapped pipe writes. However, the Windows NT 3.5 redirector allows only 17 outstanding overlapped pipe reads at any given time.

If the client uses overlapped I/O through the redirector, it is possible for the client to become deadlocked. You may need to increase the number of threads that the redirector uses for I/O; the same thing is true for the server. If your application is doing a lot of I/O, you can avoid this deadlock by creating extra threads and having them use non-overlapped I/O.

MORE INFORMATION

================

Increasing the workstation services MaxThreads parameter increases the number of kernel threads that the redirector will create, thus allowing more operations to be outstanding at any given time.

This parameter is located in:

```
HKEY_LOCAL_MACHINE\SYSTEM\
   CurrentControlSet\
   Services\
   LanmanWorkstation\
   Parameters\
   MaxThreads
```

The parameter can be set from 0 to 255 (the default is 17).

Additional reference words: 3.10 3.50

KBCategory: kbprg

KBSubcategory: BseFileio

Line Continuation and Carriage Returns in String Tables

PSS ID Number: Q44385

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

The RC compiler does not offer a line-continuation symbol for strings in string tables.

To force a carriage return into a long line of text, use one of the methods described below.

One method is to force the carriage return using \012\015. The following example demonstrates the use of \012\015 and should be considered to be on one continuous line:

```
STRINGTABLE
BEGIN
IDSLONGSTRING, "This is a long line of text so I would like \012\015
to force a carriage return."
END
```

For more information on this method, query in the Microsoft Knowledge Base on the following words:

```
STRINGTABLE WinLoadString
```

Another method of forcing a carriage return is to press ENTER and continue the line on the next line. The following example will force a carriage return after the word "like."

```
STRINGTABLE
BEGIN
IDSLONGSTRING, "This is a long line of text so I would like
to force a carriage return"
END
```

If you try to use the \n or other \ characters, the RC compiler will ignore them.

NOTE: There is a 255-character limit (per string) in a string table. For more information on this limit, please query on the following words in the Microsoft Knowledge Base:

```
STRINGTABLE length 255
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsRc

List All NetBIOS Names on a Lana

PSS ID Number: Q124960

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1 and 3.5

SUMMARY

=======

You can get a list of NetBIOS names for a lana by using the Adapter Status NetBIOS request and using the "*" character as the call name. However, on Windows NT, this method lists only the names added by the current process.

If you want to list all of the NetBIOS names on the lana, use a unique local name as the call name. This method causes the Adapter Status to be treated as a remote call, which will disable the "filtering" of names added by other processes. The sample code below demonstrates this technique.

SAMPLE CODE

```
/* The following makefile may be used to build this sample:

 !include <ntwin32.mak>

 PROJ = test.exe
 DEPS = test.obj
 LIBS_EXT = netapi32.lib

 .c.obj:
   $(cc) /YX $(cdebug) $(cflags) $(cvars) $<

 $(PROJ) : $(DEPS)
   $(link) @<<
 $**
 -out:$@
 $(conlibs)
 $(conlflags)
 $(ldebug)
 $(LIBS_EXT)
 <<

*/

#include <windows.h>
#include <stdio.h>
#include <string.h>

/*
 * LANANUM and LOCALNAME should be set as appropriate for
 * your system
 */
```

```
#define LANANUM     0
#define LOCALNAME   "MAKEUNIQUE"

#define NBCheck(x)  if (NRC_GOODRET != x.ncb_retcode) { \
                        printf("Line %d: Got 0x%x from NetBios()\n", \
                               __LINE__, x.ncb_retcode); \
                    }


void MakeNetbiosName (char *achDest, LPCSTR szSrc);
BOOL NBAddName (int nLana, LPCSTR szName);
BOOL NBReset (int nLana, int nSessions, int nNames);
BOOL NBListNames (int nLana, LPCSTR szName);
BOOL NBAdapterStatus (int nLana, PVOID pBuffer, int cbBuffer,
                      LPCSTR szName);

void
main ()
{
    if (!NBReset (LANANUM, 20, 30))
        return;

    if (!NBAddName (LANANUM, LOCALNAME))
        return;

    if (!NBListNames (LANANUM, LOCALNAME))
        return;

    printf ("Succeeded.\n");
}

BOOL
NBReset (int nLana, int nSessions, int nNames)
{
    NCB ncb;

    memset (&ncb, 0, sizeof (ncb));
    ncb.ncb_command = NCBRESET;
    ncb.ncb_lsn = 0;            /* Allocate new lana_num resources */
    ncb.ncb_lana_num = nLana;
    ncb.ncb_callname[0] = nSessions;  /* max sessions */
    ncb.ncb_callname[2] = nNames;  /* max names */

    Netbios (&ncb);
    NBCheck (ncb);

    return (NRC_GOODRET == ncb.ncb_retcode);
}

BOOL
NBAddName (int nLana, LPCSTR szName)
{
    NCB ncb;

    memset (&ncb, 0, sizeof (ncb));
```

```
    ncb.ncb_command = NCBADDNAME;
    ncb.ncb_lana_num = nLana;

    MakeNetbiosName (ncb.ncb_name, szName);

    Netbios (&ncb);
    NBCheck (ncb);

    return (NRC_GOODRET == ncb.ncb_retcode);
}

/*
 * MakeNetbiosName - Builds a name padded with spaces up to
 * the length of a NetBIOS name (NCBNAMSZ).
 */
void
MakeNetbiosName (char *achDest, LPCSTR szSrc)
{
    int cchSrc;

    cchSrc = lstrlen (szSrc);
    if (cchSrc > NCBNAMSZ)
        cchSrc = NCBNAMSZ;

    memset (achDest, ' ', NCBNAMSZ);
    memcpy (achDest, szSrc, cchSrc);
}

BOOL
NBListNames (int nLana, LPCSTR szName)
{
    int cbBuffer;
    ADAPTER_STATUS *pStatus;
    NAME_BUFFER *pNames;
    int i;

    // Allocate the largest buffer we might need
    cbBuffer = sizeof (ADAPTER_STATUS) + 255 * sizeof (NAME_BUFFER);
    pStatus = (ADAPTER_STATUS *) HeapAlloc (GetProcessHeap (), 0,
                                            cbBuffer);
    if (NULL == pStatus)
        return FALSE;


    if (!NBAdapterStatus (nLana, (PVOID) pStatus, cbBuffer, szName))
    {
        HeapFree (GetProcessHeap (), 0, pStatus);
        return FALSE;
    }

    // The list of names immediately follows the adapter status
    // structure.
    pNames = (NAME_BUFFER *) (pStatus + 1);

    for (i = 0; i < pStatus->name_count; i++)
```

```
        printf ("\t%.*s\n", NCBNAMSZ, pNames[i].name);

    HeapFree (GetProcessHeap (), 0, pStatus);

    return TRUE;
}

BOOL
NBAdapterStatus (int nLana, PVOID pBuffer, int cbBuffer, LPCSTR szName)
{
    NCB ncb;

    memset (&ncb, 0, sizeof (ncb));
    ncb.ncb_command = NCBASTAT;
    ncb.ncb_lana_num = nLana;

    ncb.ncb_buffer = (PUCHAR) pBuffer;
    ncb.ncb_length = cbBuffer;

    MakeNetbiosName (ncb.ncb_callname, szName);

    Netbios (&ncb);
    NBCheck (ncb);

    return (NRC_GOODRET == ncb.ncb_retcode);
}
```

Additional reference words: 3.10 3.50
KBCategory: kbnetwork kbnetwork
KBSubcategory: NtwkNetios

List of Articles for Win32 Base Programming Issues

PSS ID Number: Q89989

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

INSTRUCTIONS

Microsoft FastTips is available 24 hours a day, 7 days a week, from a touch-tone telephone. To order items from this catalog, first select the items you want to receive, noting the five- to six-digit number of the Item ID shown below for each item, and then:

- Dial the toll-free FastTips number (800) 936-4300.
- When prompted, select the Win32 Software Development Kit.
- Press one (1) on your phone keypad to select Express Order Service.
- When prompted, select the delivery method, fax.
- When prompted, enter your three-digit area code and seven-digit fax number on your phone keypad.
- When prompted, enter the number of the Item ID and press #, for up to five items.

When finished, simply hang up. If you have problems receiving a fax, please call (206) 635-3105.

ARTICLE LISTING

End of listing.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbref kbtlc
KBSubcategory: BseMisc

List of Articles for Win32 GDI Programming Issues

PSS ID Number: Q80828

The information in this article applies to:

- FastTips for the Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

INSTRUCTIONS

Microsoft FastTips is available 24 hours a day, 7 days a week, from a touch-tone telephone. To order items from this catalog, first select the items you want to receive, noting the five- to six-digit number of the Item ID shown below for each item, and then:

- Dial the toll-free FastTips number (800) 936-4300.
- When prompted, select the Win32 Software Development Kit.
- Press one (1) on your phone keypad to select Express Order Service.
- When prompted, select the delivery method, fax.
- When prompted, enter your three-digit area code and seven-digit fax number on your phone keypad.
- When prompted, enter the number of the Item ID and press #, for up to five items.

When finished, simply hang up. If you have problems receiving a fax, please call (206) 635-3105.

ARTICLE LISTING

ITEM ID	ARTICLE TITLE	PAGES
Q 11915	Printing in Windows Without Form Feeds	1
Q 12071	Maximum Brush Size	1
Q 22242	PRB: Area Around Text and Remainder of Window Different Colors	2
Q 22553	Displaying on the Screen What Will Print	4
Q 24179	PRB: Dotted Line Displays as Solid Line	1
Q 27225	Determining Available RGB Values of an Output Device	2
Q 27585	Specifying Windows "Bounding Box" Coordinates	1
Q 29240	Converting Colors Between RGB and HLS (HBS)	3
Q 33096	Drawing Outside a Window's Client Area	1
Q 34614	Creating Lines with a Nonstandard Pattern	2
Q 41464	Background Colors Affect BitBlt() from Mono to Color	1
Q 64520	Printing Monochrome and Color Bitmaps from Windows	1
Q 66532	Use of NULL_PEN, NULL_BRUSH, and HOLLOW_BRUSH	1
Q 67883	How to Use a DIB Stored as a Windows Resource	2
Q 69885	SetBkColor() Does Not Support Dithered Colors	2
Q 71227	Inverting Color Inverts Palette Index, Not RGB Value	2
Q 71229	Windows Regions Do Not Scale	1

End of listing.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbref kbtlc
KBSubcategory: GdiMisc

List of Articles for Win32 SDK Networking Issues

PSS ID Number: Q81246

The information in this article applies to:

- FastTips for the Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

INSTRUCTIONS
============

Microsoft FastTips is available 24 hours a day, 7 days a week, from a touch-tone telephone. To order items from this catalog, first select the items you want to receive, noting the five- to six-digit number of the Item ID shown below for each item, and then:

- Dial the toll-free FastTips number (800) 936-4300.
- When prompted, select the Win32 Software Development Kit.
- Press one (1) on your phone keypad to select Express Order Service.
- When prompted, select the delivery method, fax.
- When prompted, enter your three-digit area code and seven-digit fax number on your phone keypad.
- When prompted, enter the number of the Item ID and press #, for up to five items.

When finished, simply hang up. If you have problems receiving a fax, please call (206) 635-3105.

ARTICLE LISTING
===============

Q 124876 Sockets Applications on Microsoft Windows Platforms 2
Q 124879 PRB: Error 1 (NRC_BUFLEN) During NetBIOS Send Call 1
Q 124960 List All NetBIOS Names on a Lana 3

Q 125700 Windows 95 Support for LAN Manager APIs 2
Q 125701 Windows 95 RPC: Supported Protocol Sequences 1
Q 125702 Windows 95 Network Programming Support 1
Q 125704 Multiprotocol Support for Windows Sockets 3
Q 126716 PRB: Poor TCP/IP Performance When Doing Small Sends 1

Q 127015 Intergraph's NFS causes WinSock APIs to return error 10093 1
Q 127870 SNMP Agent Breaks Up Variable Bindings List 1
Q 127902 Where to Get the Microsoft SNMP Headers and Libraries 1
Q 128729 How to Add an SNMP Extension Agent to the NT Registry 2
Q 129022 DLC Information on LLC_DIR_SET_MULTICAST_ADDRESS Command 2

Q 129063 PRB: SnmpMgrStrToOid Assumes Oid Is in Mgmt Subtree 1
Q 129065 PRB: Getsockopt() Returns IP Address 0.0.0.0 for UDP 2
Q 129240 PRB: Building SDK SNMP Samples Results in Unresolved Externals 1
Q 129315 How to Use WinSock to Enumerate Addresses 3
Q 129316 Tips for Writing Windows Sockets Apps That Use AF_NETBIOS 3

Q 129317 Client Service For Novell Netware Doesn't Support Named Pipes 1
Q 129975 Registering Multiple RPC Server Interfaces 1
Q 130024 How Database WinSock APIs Are Implemented in Windows NT 3.5 1
Q 130562 PRB: SNMP Extension Agent Gives Exception on Windows NT 3.51 1
Q 130564 PRB: SnmpMgrGetTrap() Fails 1

Q 130942 PRB: WSAStartup() May Return WSAVERNOTSUPPORTED on Second Call 1
Q 131159 FIX: Winsock Over Appletalk (DDP) Leaks Memory 1
Q 131458 How to Broadcast Messages Using NetMessageBufferSend() 1
Q 131495 RPC CALLBACK Attribute and Unsupported Protocol Sequences 1
Q 131505 How to Set Up and Run the RNR Sample Included in the Win32 SDK 2

Q 131623 When to Use Synchronous Socket Handles & Blocking Hooks 1

End of listing.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbref kbtlc
KBSubcategory: NtwkMisc

List of Articles for Win32 SDK Samples

PSS ID Number: Q132151

The information in this article applies to:

- FastTips for the Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

INSTRUCTIONS

Microsoft FastTips is available 24 hours a day, 7 days a week, from a touch-tone telephone. To order items from this catalog, first select the items you want to receive, noting the five- to six-digit number of the Item ID shown below for each item, and then:

- Dial the toll-free FastTips number (800) 936-4300.
- When prompted, select the Win32 Software Development Kit.
- Press one (1) on your phone keypad to select Express Order Service.
- When prompted, select the delivery method, fax.
- When prompted, enter your three-digit area code and seven-digit fax number on your phone keypad.
- When prompted, enter the number of the Item ID and press#, for up to five items.

When finished, simply hang up. If you have problems receiving a fax, please call (206) 635-3105.

ARTICLE LISTING

ITEM ID	ARTICLE TITLE	PAGES
Q 11787	SAMPLE: Using Blinking Text in an Application	1
Q 40959	SAMPLE: FASTBLT Implements Smooth Movement of a Bitmap	1
Q 66652	SAMPLE: Setting Tab Stops in a Windows List Box	2
Q 66942	SAMPLE: Changing Text Alignment in an Edit Control Dynamiclly	2
Q 94326	SAMPLE: 16 and 32 Bits-Per-Pel Bitmap Formats	2
Q 112639	SAMPLE: SCLBLDLG - Demonstrates Scaleable Controls in Dialog	1
Q 118327	SAMPLE: ServerEnumDialog DLL	2
Q 118983	SAMPLE: RASberry - an RAS API Demonstration	1
Q 122787	SAMPLE: How to Use File Associations	4
Q 123605	SAMPLE: RESIZE App Shows How to Resize a Window in Jumps	2
Q 125587	SAMPLE: How to Simulate Multiple-Selection TreeView Control	2
Q 125683	SAMPLE: Customizing the TOOLBAR Control	3
Q 127071	SAMPLE: MFCOGL a Generic MFC OpenGL Code Sample	3
Q 128091	SAMPLE: How to Use Paths to Create Text Effects	3
Q 128122	SAMPLE: Implementing Multiple Threads in an OpenGL Application	2
Q 130459	SAMPLE: Adding TrueType, Raster, or Vector Fonts to System	1
Q 130476	SAMPLE: Simulating Palette Animation on Non-Palette Displays	1

Q 130804 SAMPLE: Fade a Bitmap Using Palette Animation 2
Q 130805 SAMPLE: Drawing to a Memory Bitmap for Faster Performance 1
Q 131024 SAMPLE: Drawing Three-Dimensional Text in OpenGL Applications 2

Q 131788 SAMPLE: Highlighting an Entire Row in a ListView Control 1

End of listing.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbref kbtlc
KBSubcategory: CodeSam

List of Articles for Win32 SDK Tools

PSS ID Number: Q89345

The information in this article applies to:

- FastTips for the Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

INSTRUCTIONS

Microsoft FastTips is available 24 hours a day, 7 days a week, from a touch-tone telephone. To order items from this catalog, first select the items you want to receive, noting the five- to six-digit number of the Item ID shown below for each item, and then:

- Dial the toll-free FastTips number (800) 936-4300.
- When prompted, select the Win32 Software Development Kit.
- Press one (1) on your phone keypad to select Express Order Service.
- When prompted, select the delivery method, fax.
- When prompted, enter your three-digit area code and seven-digit fax number on your phone keypad.
- When prompted, enter the number of the Item ID and press #, for up to five items.

When finished, simply hang up. If you have problems receiving a fax, please call (206) 635-3105.

ARTICLE LISTING

ITEM ID	ARTICLE TITLE	PAGES
Q 21569	RW2002 Error "Cannot Reuse String Constants" in RC.EXE	1
Q 32019	PRB: Dialog Editor Does Not Modify RC File Dialog Box Resource	1
Q 40958	PRB: DIALOG.EXE Reads Compiled .RES Files, Not .DLG Files	1
Q 44385	Line Continuation and Carriage Returns in String Tables	1
Q 64050	Use Uppercase "K" for Keywords in Windows Help Files	1
Q 67895	Multiple Columns of Text in Windows Help Files	2
Q 69081	Height and Width Limitations with Windows SDK Font Editor	1
Q 71761	PRB: Search Button Disabled in Windows Help	1
Q 74264	PRB: Dialog Editor Does Not Retain Unsupported Styles	1
Q 74278	PRB: Spy Repeatedly Lists a Single Message	1
Q 74937	Authoring Windows Help Files for Performance	2
Q 75111	Changing Hypertext Jump Color in Windows Help	2
Q 76534	Conditionally Activating a Button in Windows Help	1
Q 77748	Nesting Quotation Marks Inside Windows Help Macros	1
Q 77841	PRB: Vertical Scroll Bars Missing from Windows Help	1
Q 80945	Using #include Directive with Windows Resource Compiler	2
Q 81233	Creating Autosized Tables with Windows Help	1

Q 108227 Changes to the MSTest WFndWndC()   1
Q 110540 PRB: Quotation Marks Missing from Compiled Help File   1

Q 114604 PRB: Double Quotes Not in Help Files Compiled From Word 6 RTF   1
Q 114605 PRB: DSKLAYT2 May Create Too Many Files on a Disk Image   2
Q 114606 How to use ExitExecRestart to Install System Files   2
Q 114609 PRB: Setup Toolkit File Copy Progress Gauge not Updated   1
Q 114610 PRB: "Out of Memory Error" in the Win32 SDK Setup Sample   1

Q 118890 Using the Call-Attributed Profiler (CAP)   2
Q 120251 How to Specify Filenames/Paths in Viewer/WinHelp Commands   1
Q 125474 Differences Between the Win32 3.5 SDK and Visual C++ 2.0   2
Q 131111 PRB: Errors When Windbg Switches Not Set for Visual C++ App   1

End of listing.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbref kbtlc
KBSubcategory: TlsMisc

List of Articles for Win32 SDK User Programming Issues

PSS ID Number: Q89372

The information in this article applies to:

- FastTips for the Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, 4.0

INSTRUCTIONS

Microsoft FastTips is available 24 hours a day, 7 days a week, from a touch-tone telephone. To order items from this catalog, first select the items you want to receive, noting the five- to six-digit number of the Item ID shown below for each item, and then:

- Dial the toll-free FastTips number (800) 936-4300.
- When prompted, select the Win32 Software Development Kit.
- Press one (1) on your phone keypad to select Express Order Service.
- When prompted, select the delivery method, fax.
- When prompted, enter your three-digit area code and seven-digit fax number on your phone keypad.
- When prompted, enter the number of the Item ID and press #, for up to five items.

When finished, simply hang up. If you have problems receiving a fax, please call (206) 635-3105.

ARTICLE LISTING

ITEM ID	ARTICLE TITLE	PAGES
Q 10841	Using Extra Fields in Window Class Structure	1
Q 11337	PRB: Dialog Box and Parent Window Disabled	1
Q 11365	Creating a List Box Without a Scroll Bar	1
Q 11570	Translating Client Coordinates to Screen Coordinates	1
Q 11590	Windows Dialog-Box Style DS_ABSALIGN	1
Q 11606	Use of Allocations w/ cbClsExtra & cbWndExtra in Windows	1
Q 11619	Panning and Scrolling in Windows	1
Q 11654	Clipboard Memory Sharing in Windows	1
Q 12118	Customizing a Pop-Up Menu	1
Q 12190	Placing a Caret After Edit-Control Text	1
Q 12273	Using SendMessage() As Opposed to SendDlgItemMessage()	1
Q 19963	BeginPaint() Invalid Rectangle in Client Coordinates	1
Q 20011	Length of STRINGTABLE Resources	1
Q 24252	SetClipboardData() and CF_PRIVATEFIRST	1
Q 24646	Captions for Dialog List Boxes	1
Q 26234	PRB: Windows REQUEST Function Not Working With Excel	1
Q 29961	Location of the Cursor in a List Box	1

Q 125678 New Dialog Styles in Windows 95 1
Q 125679 New Window Styles in Windows 95 2
Q 125680 How to Subclass a Window in Windows 95 2
Q 125681 How to Calculate Dialog Base Units with Non-System-Based Font 2

Q 125682 How to Use the Small Icon in Windows 95 1
Q 125684 How to Use SS_GRAYRECT SS_BLACKRECT SS_WHITERECT in Windows 95 1
Q 125686 How to Add Windows 95 Controls to Visual C++ 2.0 Dialog Editor 1
Q 125687 PRB: Inter-thread SetWindowText() Fails to Update Window Text 1
Q 125694 How to Find Out Which Listview Column Was Right-Clicked 2

Q 125695 SystemParametersInfo() Add-On Gets or Sets System Parameters 1
Q 125703 Windows 95 Support for Network DDE 1
Q 125705 Application Version Marking in Windows 95 2
Q 125706 Customizing the FileOpen Common Dialog in Windows 95 2
Q 125752 How to Increase Windows NT System and Desktop Heap Sizes 1

Q 126625 How to Change the International Settings Programmatically 1
Q 126630 Resource Sections are Read-only 1
Q 127066 Advanced Graphics Settings Slider under Windows 95 1
Q 127190 How to Toggle the NUM LOCK, CAPS LOCK, and SCROLL LOCK Keys 1
Q 127861 Network DDE For 16-bit Windows-based Apps Under Windows NT 1

Q 128110 PRB: CBS_SIMPLE ComboBox Repainting Problem 2
Q 128125 Trusted DDE Shares 2
Q 128558 Rich Edit Control Does Not Support Unicode 1
Q 129595 WM_SYSCOLORCHANGE Must Be Sent to Windows 95 Common Controls 1
Q 129859 Using Text Bullets in a Rich Edit Control 1

Q 129860 Using Built-In Printing Features from a Rich Edit Control 3
Q 130692 PRB: Editing Labels in a TreeView Gives WM_COMMAND|IDOK Errors 1
Q 130693 How to Use CTL3D Under the Windows 95 Operating System 1
Q 130758 How to Change Small Icon for FileOpen and Other Common Dialogs 2
Q 130759 EM_SETHANDLE and EM_GETHANDLE Messages Not Supported 1

Q 130760 PRB: Can't Remove Minimize or Maximize Button from Caption Bar 1
Q 130761 Using FileOpen Common Dialog w/ OFN_ALLOWMULTIPLESELECT Style 1
Q 130762 How to Use DWL_MSGRESULT in Property Sheets & Wizard Controls 1
Q 130763 How to Create 3D Controls in Client Area of Non-Dialog Window 1
Q 130764 How to Obtain Fonts, ToolTips, and Other Non-Client Metrics 1

Q 130765 PRB: Property Sheet w/ Multiline Edit Control Ignores ESC Key 2
Q 130951 PRB: Private Button Class Can't Get BM_SETSTYLE in Windows 95 1
Q 130952 WM_CTLCOLORxxx Message Changes for Windows 95 1
Q 131025 PRB: NetDDE Fails to Connect Under Windows 95 1
Q 131225 PRB: CFileDialog::DoModal() Does Not Display FileOpen Dialog 1

Q 131259 How to Detect Slow CPU & Unaccelerated Video Under Windows 95 1
Q 131278 Using cChildren Member of TV_ITEM to Add Speed & Use Less RAM 3
Q 131279 PRB: SelectObject() Fails After ImageList_GetImageInfo() 1
Q 131280 PRB: LoadCursor() Fails on IDC_SIZE/IDC_ICON 1
Q 131281 PRB: Calling LoadMenuIndirect() with Invalid Data Hangs System 1

Q 131282 How to Display Old-Style FileOpen Common Dialog in Windows 95 1
Q 131283 PRB: Can't Use TAB to Move from Standard Controls to Custom 1

Q 131284 How to Select a Listview Item Programmatically in Windows 95   2
Q 131285 How to Use LVIF_DI_SETITEM on an LVN_GETDISPINFO Notification   2
Q 131286 PRB: LB_DIR with Long Filenames Returns LB_ERR in Windows 95   2

Q 131287 Treeviews Share Image Lists by Default   1
Q 131288 PRB: RegisterClass()/ClassEx() Fails If cbWndExtra > 40 Bytes   1
Q 131381 PRB: RichEdit Control Hides Mouse Pointer (Cursor)   1
Q 131462 How to Handle FNERR_BUFFERTOOSMALL in Windows 95   3
Q 131500 How to Obtain Icon Information from an .EXE in Windows 95   2

Q 131845 How to Modify the Width of the Drop Down List in a Combo Box   3

End of listing.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbref kbtlc
KBSubcategory: UsrMisc

List of Articles for Win32s Programming Issues

PSS ID Number: Q93064

The information in this article applies to:

 - FastTips for Microsoft Win32s versions 1.1, 1.2, and 1.25a
---

INSTRUCTIONS
============

Microsoft FastTips is available 24 hours a day, 7 days a week, from a touch-tone telephone. To order items from this catalog, first select the items you want to receive, noting the five- to six-digit number of the Item ID shown below for each item, and then:

- Dial the toll-free FastTips number (800) 936-4300.
- When prompted, select the Win32 Software Development Kit.
- Press one (1) on your phone keypad to select Express Order Service.
- When prompted, select the delivery method, fax.
- When prompted, enter your three-digit area code and seven-digit fax number on your phone keypad.
- When prompted, enter the number of the Item ID and press #, for up to five items.

When finished, simply hang up. If you have problems receiving a fax, please call (206) 635-3105.

ARTICLE LISTING
===============

End of listing.

Additional reference words: 1.10 1.20
KBCategory: kbref kbtlc
KBSubcategory: W32s

Listing Account Privileges

PSS ID Number: Q119669

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

When a user starts a process, that process takes on the security attributes of the user. The security attributes inherited from the user include privileges, which control access to system services.

MORE INFORMATION
================

To list the privileges belonging to a process (and thus to the current user), perform the following steps:

1. Call GetCurrentProcess() to obtain a handle to the current process.

2. Call GetProcessToken() to obtain the process' access token.

3. Call GetTokenInformation() to obtain the list of privileges (among other information).

4. Step through the list of privileges, using LookupPrivilegeName() and LookupPrivilegeDisplayName() to obtain the names for the program to display.

The following sample code lists the displayable privilege names and the privilege names as defined in the WINNT.H header file:

Sample Code

```
#include <windows.h>
#include <stdio.h>

void main()
{
   HANDLE hProcess, hAccessToken;
   UCHAR InfoBuffer[1000];
   PTOKEN_PRIVILEGES ptgPrivileges = (PTOKEN_PRIVILEGES)InfoBuffer;
   DWORD dwInfoBufferSize;
   DWORD dwPrivilegeNameSize;
   DWORD dwDisplayNameSize;
   UCHAR ucPrivilegeName[500];
   UCHAR ucDisplayName[500];
   DWORD dwLangId;
```

```
      UINT x;

      hProcess = GetCurrentProcess();

      OpenProcessToken( hProcess, TOKEN_READ, &hAccessToken );

      GetTokenInformation( hAccessToken, TokenPrivileges, InfoBuffer,
         sizeof(InfoBuffer), &dwInfoBufferSize);

      printf( "Account privileges: \n\n" );
      for( x=0; x<ptgPrivileges->PrivilegeCount; x++ )
      {
         dwPrivilegeNameSize = sizeof( ucPrivilegeName );
         dwDisplayNameSize = sizeof( ucDisplayName );
         LookupPrivilegeName( NULL, &ptgPrivileges->Privileges[x].Luid,
            ucPrivilegeName, &dwPrivilegeNameSize );
         LookupPrivilegeDisplayName( NULL, ucPrivilegeName,
            ucDisplayName, &dwDisplayNameSize, &dwLangId );
         printf( "%40s (%s)\n", ucDisplayName, ucPrivilegeName );
      }
   }
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Listing the Named Shared Objects

PSS ID Number: Q105764

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

Included with the Win32 SDK is an object viewer utility called WinObj that be used to list named objects, devices, dynamic-link libraries (DLLs), and so forth. To find objects such as pipes, memory, and semaphores, start WinObj and select the folder BaseNamedObjects.

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsMisc

Localizing the Setup Toolkit for Foreign Markets

PSS ID Number: Q92523

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, version 3.5

There are no localized versions of the Setup Toolkit .DLLs available. However, the resource files for these .DLLs are provided with the Windows 3.1 Software Development Kit (SDK). They can be found in the INTLDLL subdirectory in the MSSETUP directory created by the SDK Setup.

The strings in the STRINGTABLES in these .RC files can be translated. The localized .RC files can then be bound to the .DLLs using the resource compiler.

Additional reference words: 3.10 3.50 MSSETUP tool kit
KBCategory: kbtool
KBSubcategory: TlsMss

Location of the Cursor in a List Box

PSS ID Number: Q29961

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

There is no way to determine which item in the list box has the cursor when the LBN_DBLCLK message is received. You must keep track of which item has the cursor as it moves among the items. When you receive the double-click message, you will know which box has the cursor.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Location of WNet* API Functions

PSS ID Number: Q102381

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

The WNet* API routines are implemented in MPR.DLL. When linking an application that uses these routines, link with the import library MPR.LIB. For a list of which APIs can be resolved with which import libraries, see the file WIN32API.CSV, which is included in the Win32 SDK and the Visual C++ 32-bit edition.

Additional reference words: 3.10 3.50 4.00 95 lnk2001
KBCategory: kbnetwork
KBSubcategory: NtwkWinnet

Long File & Path Names Shorten When Launched Using Association

PSS ID Number: Q142275

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS

When an application is installed along a path that contains a space (as in C:\Program Files\MyApp) and you double-click a data file registered to MyApp.exe in Explorer (or any other shell), you will only get a short file name on the command line.

RESOLUTION

To resolve the problem, place quotation marks around the around the arguments list in the registry entries, as in this example:

```
HKEY_CLASSES_ROOT\MyApp\Shell\Open\Command =
  <path to MyApp.exe> "%1"
```

Without the quotation marks, short names are the result. With quotation marks, long file names are the result. This works correctly in Windows 95 and Windows NT 3.51.

STATUS

This behavior is by design.

MORE INFORMATION

The 8.3 file name specification didn't allow spaces in filenames, so the system automatically converts long file names to short filenames when no quotation marks are used to differenciate between a long file name and a list of short names seperated by spaces.

Basically the shell looks at the .exe type of the program refered to, if it is a win32 .exe, the shell passes a long name by default (as %1), if it is not, the shell passes the short name. The code that sniffs the .exe type requires the quotation marks to get this right.

Additional reference words: 4.00
KBCategory: kbenv kbprg kbprb
KBSubcategory:

Long Filenames on Windows NT FAT Partitions

PSS ID Number: Q115236

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

SUMMARY

Windows NT, version 3.5, introduces the ability to create or open files with long filenames (LFN) on Windows NT file allocation table (FAT) partitions.

UNICODE is stored on disk, so that the original name is always preserved for extended characters regardless of which code page happens to be active when reading from or writing to the disk.

The legal character set is that of the Windows NT file system (NTFS) (except for the ":" for opening NTFS alternative file streams), so you can copy arbitrary files between NTFS and FAT without losing any of the filename information.

MORE INFORMATION

The LFNs are not available from the MS-DOS DIR command, but they are available from the Windows NT DIR command. When you create an LFN on a Windows NT FAT partition, an accompanying short name is created just as on an NTFS partition. You can access the file or directory with either the long names or the short names under Windows NT.

For example, use the Microsoft Editor (MEP) to create a file named as follows on a FAT partition under Windows NT:

```
longfilename.fat
```

This is exactly how the filename appears when you run the DIR command from the Windows NT command prompt. However, when you boot the machine into MS-DOS and run the DIR command, the filename appears as follows:

```
longfi~1.fat
```

NOTE: NTFS partitions are not available under MS-DOS, so you cannot perform this experiment using an NTFS partition.

The same result can also be achieved programmatically. Build and run the following sample code on Windows NT:

Sample Code

```
#include <windows.h>

void main( )
{
   WIN32_FIND_DATA fd;
   char buf[80];

   FindFirstFile( "long*", &fd );
   wsprintf( buf, "File name is %s", fd.cFileName );
   MessageBox( NULL, buf, "Test", MB_OK );
   wsprintf( buf, "Alternate file name is %s", fd.cAlternateFileName );
   MessageBox( NULL, buf, "Test", MB_OK );
}
```

The first message box will read:

```
File name is longfilename.fat
```

The second message box will read:

```
Alternate file name is longfi~1.fat
```

Additional reference words: 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

Looking Up the Current User and Domain

PSS ID Number: Q111544

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

Program Manager displays the logged in user account and domain name in its windows title. This information can be retrieved programmatically by extracting the user SID from the current access token and then looking up the account and domain name via the LookupAccountSid() Win32 API. Below is sample code demonstrating this technique:

Sample Code

```
void ShowUserDomain(void)
{
     HANDLE hProcess, hAccessToken;
       UCHAR InfoBuffer[1000],szAccountName[200], szDomainName[200];
       PTOKEN_USER pTokenUser = (PTOKEN_USER)InfoBuffer;
       DWORD dwInfoBufferSize,dwAccountSize = 200, dwDomainSize = 200;
       SID_NAME_USE snu;

       hProcess = GetCurrentProcess();

       OpenProcessToken(hProcess,TOKEN_READ,&hAccessToken);

       GetTokenInformation(hAccessToken,TokenUser,InfoBuffer,
            1000, &dwInfoBufferSize);

       LookupAccountSid(NULL, pTokenUser->User.Sid, szAccountName,
            &dwAccountSize,szDomainName, &dwDomainSize, &snu);

    printf("%s\\%s\n",szDomainName,szAccountName);
}
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

MAKEINTATOM() Does Not Return a Valid LPSTR

PSS ID Number: Q61980

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The LPSTR returned from MAKEINTATOM() cannot be treated as a general-purpose string pointer. Instead, either use it with the Atom family or convert it into a valid string before passing it off as one.

MORE INFORMATION

The MAKEINTATOM() macro is documented on Page 370 of the "Microsoft Windows Software Development Kit Programmer's Reference" versions 2.x as returning a value of type LPSTR. This is correct, but misleading. The LPSTR value returned is a "fabricated" LPSTR and cannot be considered a general-purpose string pointer. Consider the definition of the MAKEINTATOM macro:

```
   #define MAKEINTATOM(i)  (LPSTR) ((DWORD)((WORD)(i)))
```

This tells the compiler to "take the integer value 'i', think of it as a WORD, zero-extend this into a DWORD, then think of this as a LPSTR." Thus, MAKEINTATOM(1Ah) returns 001Ah. This obviously is not the same as "1A", which would be ASCII(1)+ASCII(A)+0.

The reason this psuedo-LPSTR works with AddAtom(), for example, is that AddAtom() looks to see if the HIWORD of the LPSTR parameter is 0 (zero). If so, AddAtom() knows that the LOWORD contains an actual integer value and it simply grabs that.

The following code samples show how problems can occur with these psuedo-LPSTRs returned from MAKEINTATOM.

Incorrect

```
ATOM AddIntAtom(int iAtom)
{
    LPSTR   szAtom;

    MessageBox(hWnd,
               (szAtom=MAKEINTATOM(iAtom)),
```

```
                "Adding Atom",
                MB_OK);
    return (AddAtom(szAtom));
}
```

The above code fragment will create and return a valid atom, but the message box will display an erroneous value.

Correct

```
ATOM AddIntAtom(int iAtom)
{
    LPSTR   szAtom;
    char    szBuf[10];

    szAtom=MAKEINTATOM(iAtom);
    sprintf(szBuf, "%d", LOWORD(szAtom));   /* Here's the trick */
    MessageBox(hWnd,
                szBuf,
                "Adding Atom",
                MB_OK);
    return (AddAtom(szAtom));
}
```

In the above example, we converted the integer value contained in the LOWORD of szAtom into a character string, then used this new character string in the MessageBox() call.

Although these code fragments illustrate the limitations of a MAKEINTATOM LPSTR, they are not very realistic because you really should use GetAtomName() to get the character string of an atom. If you have not yet created an atom out of an integer value, you could just format the integer into character string directly, as follows:

```
   sprintf (szBuf, "%d", iAtom);
   MessageBox (hWnd, szBuf,....);
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Making a List Box Item Unavailable for Selection

PSS ID Number: Q74792

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In the Microsoft Windows graphical environment, an application can use a list box to enumerate options. However, there are circumstances in which one or more options may not be appropriate. The application can change the appearance of items in a list box and prevent the user from selecting one of these items by using the techniques discussed below.

MORE INFORMATION

Changing the Appearance of a List Box Item

To dim (gray) a particular item in a list box, use an owner-draw list box as follows:

1. Create a list box that has the LBS_OWNERDRAW and LBS_HASSTRINGS styles.

2. Use the following code to process the WM_MEASUREITEM message:

```
case WM_MEASUREITEM:
   ((MEASUREITEMSTRUCT FAR *)(lParam))->itemHeight = wItemHeight;
   break;
```

   wItemHeight is the height of a character in the list box font.

3. Use the following code to process the WM_DRAWITEM message:

```
#define PHDC (pDIS->hDC)
#define PRC  (pDIS->rcItem)

DRAWITEMSTRUCT FAR *pDIS;

   ...

case WM_DRAWITEM:
   pDIS = (DRAWITEMSTRUCT FAR *)lParam;
```

```
/* Draw the focus rectangle for an empty list box or an
   empty combo box to indicate that the control has the
   focus
 */
if ((int)(pDIS->itemID) < 0)
{
   switch(pDIS->CtlType)
   {
      case ODT_LISTBOX:
         if ((pDIS->itemAction) & (ODA_FOCUS))
            DrawFocusRect (PHDC, &PRC);
         break;

      case ODT_COMBOBOX:
         if ((pDIS->itemState) & (ODS_FOCUS))
            DrawFocusRect (PHDC, &PRC);
         break;
   }
   return TRUE;
}

/* Get the string */
switch(pDIS->CtlType)
{
   case ODT_LISTBOX:
      SendMessage ( pDIS->hwndItem,
                    LB_GETTEXT,
                    pDIS->itemID,
                    (LPARAM)(LPSTR)szBuf);
      break;

   case ODT_COMBOBOX:
      SendMessage ( pDIS->hwndItem,
                    CB_GETLBTEXT,
                    pDIS->itemID,
                    (LPARAM)(LPSTR)szBuf);
      break;
}

if (*szBuf == '!')   // This string is disabled
{
   hbrGray = CreateSolidBrush (GetSysColor
                                  (COLOR_GRAYTEXT));
   GrayString ( PHDC,
                hbrGray,
                NULL,
                (LPARAM)(LPSTR)(szBuf + 1),
                0,
                PRC.left,
                PRC.top,
                0,
                0);
   DeleteObject (hbrGray);

   /* SPECIAL CASE - Need to draw the focus rectangle if
```

```
      there is no current selection in the list box, the
      1st item in the list box is disabled, and the 1st
      item has gained or lost the focus
    */
   if (pDIS->CtlType == ODT_LISTBOX)
   {
      if (SendMessage ( pDIS->hwndItem,
                        LB_GETCURSEL,
                        0,
                        0L) == LB_ERR)
         if ( (pDIS->itemID == 0) &&
              ((pDIS->itemAction) & (ODA_FOCUS)))
            DrawFocusRect (PHDC, &PRC);
   }
}

else  // This string is enabled
{
   if ((pDIS->itemState) & (ODS_SELECTED))
   {
      /* Set background and text colors for selected
         item */
      crBack = GetSysColor (COLOR_HIGHLIGHT);
      crText = GetSysColor (COLOR_HIGHLIGHTTEXT);
   }
   else
   {
      /* Set background and text colors for unselected
         item */
      crBack = GetSysColor (COLOR_WINDOW);
      crText = GetSysColor (COLOR_WINDOWTEXT);
   }

   // Fill item rectangle with background color
   hbrBack = CreateSolidBrush (crBack);
   FillRect (PHDC, &PRC, hbrBack);
   DeleteObject (hbrBack);

   // Set current background and text colors
   SetBkColor (PHDC, crBack);
   SetTextColor (PHDC, crText);

   // TextOut uses current background and text colors
   TextOut ( PHDC,
             PRC.left,
             PRC.top,
             szBuf,
             lstrlen(szBuf));

   /* If enabled item has the input focus, call
      DrawFocusRect to set or clear the focus
      rectangle */
   if ((pDIS->itemState) & (ODS_FOCUS))
      DrawFocusRect (PHDC, &PRC);
}
```

```
      return TRUE;
```

Strings that start with "!" are displayed dimmed. The exclamation mark character is not displayed.

Preventing Selection

To prevent a dimmed string from being selected, create the list box with the LBS_NOTIFY style. Then use the following code in the list box's parent window procedure to process the LBN_SELCHANGE notification:

```
  case WM_COMMAND:

    switch (wParam)
      {

       ...

      case IDD_LISTBOX:
        if (LBN_SELCHANGE == HIWORD(lParam))
          {
          idx = (int)SendDlgItemMessage(hDlg, wParam,
                        LB_GETCURSEL, 0, 0L);
          SendDlgItemMessage(hDlg, wParam, LB_GETTEXT, idx,
                        (LONG)(LPSTR)szBuf);
          if ('!' == *szBuf)
            {
            // Calculate an alternate index here
            // (not shown in this example).

            // Then set the index.
            SendDlgItemMessage(hDlg, wParam, LB_SETCURSEL, idx, 0L);
            }
          }
        break;

       ...

      }
   break;
```

When the user attempts to select a dimmed item, the alternate index calculation moves the selection to an available item.

Additional reference words: 3.00 3.10 3.50 4.00 95 listbox
KBCategory: kbui
KBSubcategory: UsrCtl

Managing Per-Window Accelerator Tables

PSS ID Number: Q82171

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

In the Windows environment, an application can have several windows, each with its own accelerator table. This article describes a simple technique requiring very little code that an application can use to translate and dispatch accelerator key strokes to several windows. The technique employs two global variables, ghActiveWindow and ghActiveAccelTable, to track the currently active window and its accelerator table, respectively. These two variables, which are used in the TranslateAccelerator function in the application's main message loop, achieve the desired result.

MORE INFORMATION
================

The key to implementing this technique is to know which window is currently active and which accelerator table, if any, is associated with the active window. To track this information, process the WM_ACTIVATE message that Windows sends each time an application gains or loses activation. When a window loses activation, set the two global variables to NULL to indicate that the window and its accelerator table are no longer active. When a window that has an accelerator table gains activation, set the global variables appropriately to indicate that the accelerator table is active. The following code illustrates how to process the WM_ACTIVATE message:

```
case WM_ACTIVATE:
   if (wParam == 0)  // indicates loss of activation
   {
      ghActiveWindow = ghActiveAccelTable = NULL;
   }
   else              // indicates gain of activation
   {
      ghActiveWindow = <this window>;
      ghActiveAccelTable = <this window's accelerator table>;
   }
   break;
```

The application's main message loop resembles the following:

```
while (GetMessage(&msg,     // message structure
                  NULL,     // handle of window receiving the msg
                  NULL,     // lowest message to examine
                  NULL))    // highest message to examine
{
   if (!TranslateAccelerator(ghActiveWindow, // active window
                             ghActiveAccelTable, // active accelerator
                             &msg))
   {
      TranslateMessage(&msg); // Translates virtual key codes
      DispatchMessage(&msg);  // Dispatches message to
                              //   window procedure
   }
}
```

Under Windows version 3.1, the WM_ACTIVATE message with the wParam set to WA_INACTIVE indicates loss of activation.

Under Win32, the low-order word of wParam set to WA_INACTIVE indicates deactivation.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrMen

Mapping .INI File Entries to the Registry

PSS ID Number: Q104136

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

Under Windows NT, .INI file variables are mapped into the Registry as defined in the

```
\HKEY_LOCAL_MACHINE
   \Software\Microsoft\WindowsNT\CurrentVersion\IniFileMapping
```

mapping key. The Win32 Profile application programming interface (API) functions look for a mapping by looking up the filename extension portion of the profile file. If a match is found, then the search continues under that node for the specified application name. If a match is found, then the search continues for the variable name. If the variable name is not found, the value of the (NULL) variable name is a string that points to a node in the Registry, whose value keys are the variable names. If a specific mapping is found for the variable name, then its value points to the Registry value that contains the variable value.

The Profile API calls go to the Windows server to look for an actual .INI file, and read and write its contents, only if no mapping for either the application name or filename is found. If there is a mapping for the filename but not the application name, and there is a (NULL) application name, the value of the (NULL) variable will be used as the location in the Registry of the variable, after appending the application name to it.

In the string that points to a Registry node, there are several prefixes that change the behavior of the .INI file mapping:

- ! - This character forces all writes to go both to the Registry and to the .INI file on disk.

- # - This character causes the Registry value to be set to the value in the Windows 3.1 .INI file when a new user logs in for the first time after setup.

- @ - This character prevents any reads from going to the .INI file on disk if the requested data is not found in the Registry.

- USR: - This prefix stands for HKEY_CURRENT_USER, and the text after the prefix is relative to that key.

- SYS: - This prefix stands for HKEY_LOCAL_MACHINE\Software, and the text after the prefix is relative to that key.

Additional reference words: 3.10 3.50 inifilemapping
KBCategory: kbprg

KBSubcategory: BseMisc

Mapping Modes and Round-Off Errors

PSS ID Number: Q89215

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Mapping modes, window extents/origins, and viewport extents/origins allow for very powerful coordinate manipulation, such as scaling or moving objects. However, you should be aware that there are cases when using mapping modes other than MM_TEXT results in improper painting due to round-off errors.

Round-off errors occur when one logical unit does not equal one device unit, and an application requests the graphics device interface (GDI) to perform an action that would result in a nonintegral number of pixels needing to be drawn, scrolled, blt'd, and so on.

Round-off errors can manifest themselves in many ways, including unpainted portions of a client area when scrolling, gaps between objects that shouldn't have gaps (or vice versa), objects that shrink or grow one pixel depending on where they are on the screen, objects of unexpected sizes, and so on.

MORE INFORMATION

To better understand round-off errors, consider the following code:

```
SetMapMode (hDC, MM_ANISOTROPIC);
SetWindowExt (hDC, 2, 2);
SetViewportExt (hDC, 3, 3);
PatBlt (hDC, 0, 0, 5, 2, BLACKNESS);
```

This code tells the GDI to treat two logical units (the coordinates used by most GDI functions), in both the vertical and horizontal direction, as being equal to three device units (pixels). It then asks the GDI to draw what amounts to a black rectangle five logical units wide by three logical units tall starting at the logical point (0,0).

The GDI would translate this request into a request to draw a rectangle 7.5 (5 * 3/2 = 7.5) pixels wide by 3 (2 * 3/2 = 3) pixels tall. However, display cards cannot draw half a pixel, so the GDI would either have to round the width up to 8 or truncate it to 7. If

an application relied on one behavior or the other, improper painting could occur.

Note that using mapping modes, window origins/extents, and viewport origins/extents does not mean that an application will have round-off errors. The occurrence of round-off errors depends on what these features are used for, the structure of the application, and other factors. Many applications take advantage of mapping modes, window origins/extents, and viewport origins/extents without ever encountering adverse round-off errors.

If an application exhibits round-off errors, there are a number of ways to prevent them, some which are described below.

Method 1

Only use MM_TEXT mapping mode, where one logical unit always equals one device unit. However, the application must do all its own scaling and moving of objects. The benefit of this approach is that the application has strict control over how objects are scaled and moved; you can look at your code to see the algebra that leads to round-off errors, and counter these errors appropriately. The drawback to this approach is that it makes the code more complicated and harder to read than it might be if the SetMapMode, SetWindowOrg, SetWindowExt, SetViewportOrg, and SetViewportExt functions were used.

Method 2

Mix MM_TEXT mapping mode with the mapping mode required. Sometimes applications only have round-off problems with certain types of objects. For example, in a graphing program, the application might want to set a certain mapping mode to draw a bar graph; this mapping mode might cause the fonts that the application draws to be of the wrong size.

To work around problems like this, mix MM_TEXT mapping mode with your mapping mode of choice. You could use MM_TEXT when dealing with objects that need exact sizes or placement and the other mapping mode for other drawing.

The benefits and drawbacks of this method are almost the same as those for method 1. However, with method 2, applications can take advantage of mapping modes for some of the scaling and moving of objects.

Method 3

If window/viewport origins/extents are set at compile time, be sure to only do operations that would result in no round-off errors. For example, take the fraction WindowExt over ViewportExt, and reduce this fraction. Then only do operations that involve multiples of the reduced WindowExt values. For example, given the following

```
    WindowExt   = ( 6, 27)
    ViewportExt = (50, 39)
```

turn this into a fraction and reduce it. It yields:

```
    in x direction:   6/50 = (2 * 3) / (2 * 5 * 5)  = 3/25
    in y direction:  27/39 = (3 * 3 * 3) / (3 * 13) = 9/13
```

Therefore, anything done in the x direction could be done using a multiple of three logical units; anything done in the y direction could be done using a multiple of nine logical units. For example, if the application wanted to scroll the window horizontally, it could scroll 3, 6, 9, 12, and so on logical units without having to deal with rounding errors. By using these values, the application will never have round-off errors.

One benefit of this method is that an application can take advantage of window origins/extents and viewport origins/extents. A disadvantage is that the application is limited to a certain set of origins/extents (that is, those built into the application at compile time).

Method 4

Applications can perform method 3 on-the-fly. This allows the application to deal with arbitrary window origins/extents and viewport origins/extents. To determine the minimum number of logical units an application could use given arbitrary extent values, the following code might prove useful (the code shown is for determining the value to use in the horizontal direction):

```
int GetMinWinXFactor (HDC hDC)
{
   int nMinX, xWinExt, xViewExt, nGCD;

   xWinExt  = LOWORD (GetWindowExt (hDC));
   xViewExt = LOWORD (GetViewportExt (hDC));
   while ((nGCD = GreatestCommonDivisor (xWinExt, xViewExt)) != 1)
      {
      xWinExt  /= nGCD;
      xViewExt /= nGCD;
      }
   return xWinExt;
}

int GreatestCommonDivisor (int i, int j)
{
   while (i != j)
      if (i > j)
         i -= j;
      else
         j -= i;
   return i;
}
```

The return value from the GetMinWinXFactor function above can then be used just like in method 3 (that is, the application can do all output based on multiples of this value).

Final Notes

The discussion above did not take into account the window origin, which can contribute to round-off errors. How origins and extents affect the coordinates that GDI uses is summed up in "Programmer's Reference.

Developers using mapping modes are encouraged to study the equations presented in the programmer's reference. The GDI uses these equations when converting between logical and device units. When round-off errors occur in an application, it is always a good idea to run the numbers through these equations to try to determine the cause of the errors.

Additional reference words: 3.00 3.10 3.50 4.00 95 rounding
KBCategory: kbgraphic
KBSubcategory: GdiDc

Maximum Brush Size

PSS ID Number: Q12071

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

The maximum size of a brush is determined by the OEM driver. In theory, any brush size can be created. However, the brushes are realized by the OEM driver. The only requirement the driver must satisfy is that it must be able to handle 8 x 8 brushes.

If the driver realizes variable sizes, passing any bitmap is acceptable. The driver decides what to do with brushes larger than 8 x 8.

Currently, sample drivers use the top left 8 x 8 piece of the pattern; however, if the OEM wants to use the whole pattern, that also is proper. The limit of the current device drivers is 8 x 8 because of a performance trade-off.

The amount of pattern realized is a display driver-dependent issue.

Additional reference words: 3.00 3.10 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiPnbr

Memory Handle Allocation

PSS ID Number: Q91194

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

This article discusses the limitations that exist when allocating memory handles.

The minimum block you can reserve with each call to VirtualAlloc() is 64K. It is a good idea to confirm this number by checking the allocation granularity returned by GetSystemInfo().

With HeapAlloc(), there is no limit to the number of handles that can be allocated. GlobalAlloc() and LocalAlloc() (combined) are limited to 65536 total handles for GMEM_MOVEABLE and LMEM_MOVEABLE memory per process. Note that this limitation does not apply to GMEM_FIXED or LMEM_FIXED memory.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMm

Memory Requirements for a Win32 App vs. the Win16 Version

PSS ID Number: Q117892

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s versions 1.1, 1.15, and 1.2

SUMMARY

A Win32 port of a Windows-based application generally requires more virtual memory than the original Windows-based application. However, it is possible for the Win32 version of the application to have a smaller working set. The working set is the certain number of pages that the virtual memory manager must keep in memory for a process to execute efficiently. If you lower the working set of an application, it will use less RAM.

MORE INFORMATION

It can appear that the Win32-based version of an application running on Win32s requires more RAM than the Windows-based version of the application running on the same machine. This is because segments of a Windows-based application are loaded only as they are referenced, while the address space is reserved for the Win32-based application and its DLLs (dynamic-link libraries) at program load. Therefore, the memory count that is displayed by many "About" boxes is misleading: for the Windows-based application, the free memory reported is reduced by the number of segments actually loaded; for the Win32-based application, the free memory reported is reduced by the total address space required. However, this free memory represents only the virtual address space that all applications share, not the amount of RAM actually used.

You can use WMem to determine the address space used, the number of physical pages of RAM used, and to get an estimate of the working set. On a machine that has enough RAM to load the whole application without swapping, run only Program Manager and WMem. Use SHIFT+double-click in WMem and write down the available physical memory. Activate the application and use SHIFT+double-click again. The difference between the available physical memory before and after activating the application is a rough estimate of the working set. Test your application further and see how the working set changes during execution.

The working set of a Win32-based application can be decreased 30 percent or more with the use of the Working Set Tuner, included in the Win32 SDK. However, a Win32-based application may fail to load on Win32s even if its working set is significantly smaller than the free RAM (for example, 100K working set versus 1 megabyte free RAM). The entire application, DLLs included, must be mapped into the virtual address space.

The virtual memory size is set by the system at boot time, based on several factors. RAM is one factor, free disk space is another. The system must be able to allocate enough space for the swap file on disk. Windows, by default, allows the size of the swap file to be a maximum of 4 times larger than available RAM. This constant (4) can be modified by setting PageOverCommit in the 386enh section of the SYSTEM.INI file. Valid settings are between 1 and 20. Setting PageOverCommit to a value larger than 4 will result in less efficient usage of resources and slower execution, but it will allow you to run applications that otherwise are not able to run.

Additional reference words: 1.10 1.20 3.10 3.50 4.00 95 ProgMan
KBCategory: kbprg
KBSubcategory: BseMm

Menu Operations When MDI Child Maximized

PSS ID Number: Q71836

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Pop-up menus added to an MDI application's menu using InsertMenu() with MF_BYPOSITION will be inserted one position further left than expected if the active MDI child window is maximized. This behavior occurs because the system menu of the active MDI child is inserted into the first position of the MDI frame window's menu bar.

To avoid this problem, if the active child is maximized when a new pop-up is inserted by position, add 1 (one) to the position value that would otherwise have been used. To determine that the currently active child window is maximized, send a WM_MDIGETACTIVE message to the MDI client window. In 16-bit Windows, if the high word of the return value from this message contains 1, the active child window is maximized. In Win32, you need to pass a pointer to a BOOL in the lParam. If the child window is maximized, then Windows will set the BOOL pointed to by the lParam to TRUE.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrMdi

Message Retrieval in a DLL

PSS ID Number: Q96479

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

When a function in a dynamic-link library (DLL) retrieves messages on behalf of a calling application, the following must be addressed:

- The application and the DLL may be re-entered.
- The application can terminate while in the DLL's message retrieval loop.
- The DLL must allow the application to preprocess any messages that may be retrieved.

MORE INFORMATION
================

The following concerns arise when a function in a DLL retrieves messages by calling GetMessage or PeekMessage:

- When the DLL function retrieves, translates, and dispatches messages, the calling application and the DLL function may be re-entered. This is because message retrieval can cause the calling application to respond to user input while waiting for the DLL function to return. The DLL function can return a reentrancy error code if this happens. To prevent reentrancy, disable windows and menu-items, or use a filter in the GetMessage or PeekMessage call to retrieve specific messages.

- The application can terminate while execution is in the DLL function's message retrieval loop. The WM_QUIT message retrieved by the DLL must be re-posted and the DLL function must return to the calling application. This allows the calling application's message retrieval loop to retrieve WM_QUIT and terminate.

- When the DLL retrieves messages, it must allow the calling application to preprocess the messages (to call TranslateAccelerator, IsDialogMessage, and so forth) if required. This is be done by using CallMsgFilter to call any WH_MSGFILTER hook that the application may have installed.

The following code shows a message retrieval loop in a DLL function that waits for a PM_COMPLETE message to signal the end of processing:

```
while (notDone)
{
   GetMessage(&msg, NULL, 0, 0);

   // PM_COMPLETE is a WM_USER message that is posted when
   // the DLL function has completed.
   if (msg.message == PM_COMPLETE)
   {
      Clean up and set result variables;
      return COMPLETED_CODE;
   }
   else if (msg.message == WM_QUIT)  // If application has terminated...
   {
      // Repost WM_QUIT message and return so that calling
      // application's message retrieval loop can exit.
      PostQuitMessage(msg.wParam);
      return APP_QUIT_CODE;
   }

   // The calling application can install a WH_MSGFILTER hook and
   // preprocess messages when the nCode parameter of the hook
   // callback function is MSGF_MYHOOK. This allows the calling
   // application to call TranslateAccelerator, IsDialogMessage, etc.
   if (!CallMsgFilter(&msg, MSGF_MYHOOK))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }

   :
   :
}
```

Define MSGF_HOOK to a value greater than or equal to MSGF_USER defined in WINDOWS.H to prevent collision with values used by Windows.

Preprocessing Messages in the Calling Application

The calling application can install a WH_MSGFILTER hook to preprocess messages retrieved by the DLL. It is not required for the calling application to install such a hook if it does not want to preprocess messages.

```
   lpfnMsgFilterProc = MakeProcInstance((FARPROC)MsgFilterHookFunc,
                                        ghInst);
   hookprocOld = SetWindowsHook(WH_MSGFILTER, lpfnMsgFilterProc);
   // Call the function in the DLL.
   DLLfunction();
   UnhookWindowsHook(WH_MSGFILTER, lpfnMsgFilterProc);
   FreeProcInstance(lpfnMsgFilterProc);
```

MsgFilterHookFunc is the hook callback function:

```
LRESULT CALLBACK MsgFilterHookFunc(int nCode, WPARAM wParam,
                                   LPARAM lParam)
{
   if (nCode < 0)
      return DefHookProc(nCode, wParam, lParam, &hookprocOld);

   // If CallMsgFilter is being called by the DLL.
   if (nCode == MSGF_MYHOOK)
   {
      Preprocess message (call TranslateAccelerator,
          IsDialogMessage etc.);
      return 0L if the DLL is to call TranslateMessage and
          DispatchMessage. Return 1L if TranslateMessage and
          DispatchMessage are not to be called.
   }
   else return 0L;
}
```

Additional reference words: 3.10 3.50 3.51 4.00 95 yield
KBCategory: kbui
KBSubcategory: UsrMsg

Method for Sending Text to the Clipboard

PSS ID Number: Q35100

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Sending text to the Clipboard is usually a cumbersome process of allocating and locking global memory, copying the text to that memory, and sending the Clipboard the memory handle. This method involves many pointers and handles and makes the entire process difficult to use and understand.

Clipboard I/O is easily accomplished with an edit control. If a portion of text is highlighted, an application can send the edit control a WM_COPY or WM_CUT message to copy or cut the selected text to the Clipboard. In the same manner, text can be pasted from the Clipboard by sending a WM_PASTE message to an edit control.

The following example demonstrates how to use an edit control transparently within an application to simplify sending and retrieving text from the Clipboard. Note that this code will not be as fast as setting or getting the Clipboard data explicitly, but it is easier from a programming standpoint, especially if the text to be sent is already in an edit control. Note also that the presence of the edit window will occupy some additional memory.

MORE INFORMATION

For simplified Clipboard I/O, do the following:

1. Declare a global HWND, hEdit, which will be the handle to the edit control.

2. In WinMain, use CreateWindow() to create a child window edit control. Use the style WS_CHILD, and give the control dimensions large enough to hold the most text that may be sent to or received from the Clipboard. CreateWindow() returns the handle to the edit control that should be saved in hEdit.

3. When a Cut or Copy command is invoked, use SetWindowText() to place the desired string in the edit control, then use SendMessage() to select the text and copy or cut it to the Clipboard.

4. When a Paste command is invoked, use SetWindowText() to clear the edit control, then use SendMessage() to paste text from the Clipboard. Finally, use GetWindowText() to copy the text in the edit control to a string buffer.

The actual coding for this procedure is as follows:

```
    .
    .
    .

    #define ID_ED    100
    HWND       hEdit;

    .
    .
    .
    /* In WinMain: hWnd is assumed to be the handle of the parent window,
*/
    /* hInstance is the instance handle of the parent.
*/
    /* The "EDIT" class name is required for this method to work. ID_ED
*/
    /* is an ID number for the control, used by Get/SetDlgItemText.
*/

    hEdit=CreateWindow("EDIT",
                       NULL,
                       WS_CHILD | BS_LEFTTEXT,
                       10, 15, 270, 10,
                       hWnd,
                       ID_ED,
                       hInstance,
                       NULL);


    .
    .
    .

    /* In the procedure receiving CUT, COPY, and PASTE commands:         */
    /* Note that the COPY and CUT cases perform the same actions, only   */
    /* the CUT case clears out the edit control.                         */

    /* Get the string length */
    short    nNumChars=strlen(szText);

    case CUT:
        /* First, set the text of the edit control to the desired string */
        SetWindowText(hEdit, szText);

        /* Send a message to the edit control to select the string */
        SendMessage(hEdit, EM_SETSEL, 0, MAKELONG(0, nNumChars));

        /* Cut the selected text to the clipboard */
```

```
        SendMessage(hEdit, WM_CUT, 0, 0L);
        break;

    case COPY:
        /* First, set the text of the edit control to the desired string */
        SetWindowText(hEdit, szText);

        /* Send a message to the edit control to select the string */
        SendMessage(hEdit, EM_SETSEL, 0, MAKELONG(0, nNumChars));

        /* Copy the text to the clipboard */
        SendMessage(hEdit, WM_COPY, 0, 0L);
        break;


    case IDM_PASTE:
        /* Check if there is text available */
        if (IsClipboardFormatAvailable(CF_TEXT))
            {
            /* Clear the edit control */
            SetWindowText(hEdit, "\0");

            /* Paste the text in the clipboard to the edit control */
            SendMessage(hEdit, WM_PASTE, 0, 0L);

            /* Get the test from the edit control into a string.    */
            /* nNumChars represents the number of characters to get */
            /* from the edit control.                               */
            GetWindowText(hEdit, szText, nNumChars);
            }
        else
            MessageBeep(0); /* Beep on illegal request */
        break;
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrClp

Microsoft Win32s Upgrade

PSS ID Number: Q122235

The information in this article applies to:

- Microsoft Win32s, version 1.30c

SUMMARY
=======

Microsoft Win32s version 1.30c is the latest version of the software that allows you to run Win32-based applications on Windows version 3.1 or Windows for Workgroups versions 3.1 and later. The Win32s upgrade is now available in the Microsoft Software Library (MSL) as PW1118.EXE.

The Microsoft Win32s upgrade includes OLE support.

The Win32s upgrade is intended as an end-user upgrade and is not intended for further redistribution.

You can find PW1118.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile PW1118.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get PW1118.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and Download PW1118.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download PW1118.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

MORE INFORMATION
================

Microsoft Win32s version 1.30c is the latest version of the software that allows you to run Win32-based applications on Windows version 3.1 or Windows for Workgroups version 3.1 and later.

Installing Win32s from PW1118.EXE

The following are instructions for installing Win32s if you have downloaded the PW1118.EXE file. The Win32s files are provided in a compressed state in the PW1118.EXE file, which is a self-extracting file.

1. Create a directory on your hard disk for the decompressed files. For example, you could create a directory named Install by entering the following at the command prompt:

   mkdir Install

2. Copy PW1118.EXE to the new Install directory, change directories to the Install directory, and type the following at the command prompt:

   pw1118

   This decompresses the Win32s files into an installation layout (setup program) in the Install directory.

To Install Win32s on Your System

1. Run Windows and start the File Manager.

2. Open the Install directory (the directory created in step 1 of the setup instructions above).

3. Double-click Setup.exe. The setup program installs Win32s and restarts the system to update the system files. Win32s is now installed or upgraded on your system, and any obsolete Win32s files have been removed.

4. Delete the Install directory and its contents.

REFERENCES

For additional information, please see the following articles in the Microsoft Knowledge Base:

   ARTICLE-ID: Q106715
   TITLE   : How to Troubleshoot Win32s Installation Problems

   ARTICLE-ID: Q120486
   TITLE   : How to Remove Win32s

   ARTICLE-ID: Q117153
   TITLE   : Display Problems with Win32s and the S3 Driver.

Additional reference words: 1.30c
KBCategory: kbsetup kbfile kbwebcontent
KBSubcategory: W32s

MIDL 1.0 and MIDL 2.0 Full Pointers Do Not Interoperate

PSS ID Number: Q115830

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.5

Microsoft Remote Procedure Call (RPC), version 1.0, has minimal support for full pointers, so the version of the MIDL compiler with Microsoft RPC version 1.0 (MIDL 1.0) treats full pointers (specified with the ptr attribute) as unique pointers (specified with the unique attribute).

The MIDL compiler with Microsoft RPC, version 2.0 (MIDL 2.0), supports full pointers. Because of the way Microsoft RPC, version 1.0, handles the on-wire representation of pointers, applications compiled using MIDL 2.0 full pointers cannot operate interactively with applications compiled using MIDL 1.0 full pointers.

The workaround is to recompile the MIDL 2.0 application to use unique pointers.

Additional reference words: 3.50
KBCategory: kbnetwork
KBSubcategory: NtwkRpc

Mirroring Main Menu with TrackPopupMenu()

PSS ID Number: Q99806

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

A developer may want to use TrackPopupMenu() to display the same menu that is used by the main window. TrackPopupMenu() takes a pop-up menu, while GetMenu() and LoadMenu() both return handles to top level menus, and therefore you cannot use the menus returned from these functions with TrackPopupMenu(). To "mirror" the main menu, you must create pop-up menus with the same strings, style, and IDs as the main menu. To do this, use the following Windows APIs:

   GetMenu()
   CreatePopupMenu()
   GetMenuState()
   GetMenuString()
   GetSubMenu()
   AppendMenu()

MORE INFORMATION

The following code displays the same menu as the main window when the right mouse button is clicked:

```
// In the main window procedure...

    case WM_RBUTTONDOWN:
    {

            HMENU hMenu;            // The handle to the main menu.
            int nMenu;              // The index of the menu item.
            POINT pt;             // The point to display the track menu.
            HMENU hMenuOurs;        // The pop-up menu that we are creating.
            UINT nID;             // The ID of the menu.
            UINT uMenuState;        // The menu state.
            HMENU hSubMenu;        // A submenu.
            char szBuf[128];        // A buffer to store the menu string.

// Get the main menu.
            hMenu = GetMenu(hWnd);
            nMenu = 0;
```

```
// Create a pop-up menu.
           hMenuOurs = CreatePopupMenu();

// Get menu state will return the style of the menu
// in the lobyte of the unsigned int. Return value
// of -1 indicates the menu does not exist, and we
// have finished creating our pop up.
           while ((uMenuState =
              GetMenuState(hMenu,nMenu,MF_BYPOSITION)) != -1)
           {
                if (uMenuState != -1)
                {
// Get the menu string.
              GetMenuString(hMenu,nMenu, szBuf,128,MF_BYPOSITION);
                     if (LOBYTE(uMenuState) & MF_POPUP) // It's a pop-up
menu.
                     {
                         hSubMenu = GetSubMenu(hMenu,nMenu);
                            AppendMenu(hMenuOurs,
                     LOBYTE(uMenuState),hSubMenu,szBuf);
                     }
                     else  // Is a menu item, get the ID.
                     {
                         nID = GetMenuItemID(hMenu,nMenu);
                         AppendMenu(hMenuOurs,LOBYTE(uMenuState),nID,szBuf);

                     }
                     nMenu++;  // Get the next item.
                }
        }
        pt = MAKEPOINT(lParam);
// TrackPopupMenu expects screen coordinates.
        ClientToScreen(hWnd,&pt);
           TrackPopupMenu(hMenuOurs,
          TPM_LEFTALIGN|TPM_RIGHTBUTTON,
          pt.x,pt.y,0,hWnd,NULL);

// Because we are using parts of the main menu in our
// pop-up menu, we can't just delete the pop-up menu, because
// that would also delete the main menu. So we must
// go through the pop-up menu and remove all the items.
        while (RemoveMenu(hMenuOurs,0,MF_BYPOSITION))
            ;

// Destroy the pop-up menu.
        DestroyMenu(hMenuOurs);
    }
    break;
```

If the menu is never dynamically modified, then the menu hMenuOurs could be made static and created inside the WM_CREATE message, and destroyed in the WM_DESTROY message.

To see how this function works, paste this code into the MENU sample

application shipped with both Microsoft Visual C/C++ and Microsoft C/C++ version 7.0 in the file MENU.C in the MenuWndProc() function.

Additional reference words: 3.10 3.50 3.51 4.00 95 popup
KBCategory: kbui
KBSubcategory: UsrMen

Missing Japanese Win32s-J Version 1.25 Files

PSS ID Number: Q130844

The information in this article applies to:

- Microsoft Japanese Win32s-J version 1.25

SUMMARY

=======

The files missing from the Japanese Win32s version 1.25 release included in MSDN (Microsoft Developers Network) Development Platform April '95 (CD 1) are now available.

Download WIN32SJ.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download WIN32SJ.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get WIN32SJ.EXE

MORE INFORMATION

================

The following files are missing from the MSDN Win32s-J v1.25 NODEBUG directory:

ole2conv.dll
ole2conv.sym
ole2disp.dll
ole2disp.sym
ole2nls.dll
ole2nls.sym
ole2prox.dll
ole2prox.sym
ole2thk.dll
ole32.dll
oleaut32.dll
oleaut32.sym
olecli.dll
olecli32.dll
olecli32.sym
olesvr32.dll
olesvr32.sym
sck16thk.dll
shell32.dll
shell32.sym
stdole.tlb

stdole32.tlb
storage.dll
typelib.dll
typelib.sym
version.dll
version.sym
w32s.386
w32scomb.dll
w32scomb.sym
w32skrnl.dll
w32sys.dll
win32s.exe
win32s16.dll
winmm.dll
winmm16.dll
winspool.drv
winspool.sym
wsock32.dll
wsock32.sym

All of these files are included in the self-extracting file (WIN32SJ.EXE) available from the MSL.

Additional reference words: 1.25 3.10 kbinf kbmsdn
KBCategory: kbother kbfile
KBSubcategory: wintldev

Mixer Manager Incorporated into NT 3.5 SDK and DDK

PSS ID Number: Q124504

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, and 4.0
- Microsoft Win32 Device Development Kit (DDK) version 3.5

In Windows version 3.1, the Mixer Manager Application Programming Interface (API) and Mixer Manager Service Provider Interface (SPI) are provided by the Windows Sound System Version 2.0 Driver Development Kit (DDK). The Windows Sound System Version 2.0 DDK supports 16-bit application and driver development for digital audio mixers.

Starting with Windows NT version 3.5, the Mixer Manager API has been incorporated into the Win32 SDK, and the Mixer Manager SPI has been incorporated into the Win32 DDK. As a result, there is no 32-bit version of the Windows Sound System Version 2.0 DDK; all of its functions have been incorporated into the Win32 SDK and DDK.

Additional reference words: 3.50 4.00 95 WSS DDK Mixer Manager msmixmgr msmixmgr.dll
KBCategory: kbmm
KBSubcategory: MMMixer

MS Setup Disklay2 Utility Calls COMPRESS.EXE Internally

PSS ID Number: Q103071

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

The Disklay2 utility that comes with the Microsoft Setup Toolkit calls the COMPRESS.EXE application internally to compress files. If this utility cannot be found during the execution of Disklay2, the infamous "Bad Command or Filename" error results. The screen dump of the results of Disklay2's progress will contain this error message. The solution is to ensure COMPRESS.EXE is available in the path.

COMPRESS.EXE is a component of the Windows Software Development Kit (SDK).

Additional reference words: 3.10 3.50 4.00 95 mssetup tool kit
KBCategory: kbtool
KBSubcategory: TlsMss

MultiByteToWideChar() Codepages CP_ACP/CP_OEMCP

PSS ID Number: Q108450

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY
=======

MultiByteToWideChar() maps a character string to a wide-character string. The declaration of this application programming interface (API) is as follows:

```
int MultiByteToWideChar(uCodePage, dwFlags, lpMultiByteStr,
   cchMultiByte, lpWideCharStr, cchWideChar)

UINT uCodePage;         /* codepage                         */
DWORD dwFlags;          /* character-type options           */
LPCSTR lpMultiByteStr;  /* address of string to map         */
int cchMultiByte;       /* number of characters in string   */
LPWSTR lpWideCharStr;   /* address of wide-character buffer */
int cchWideChar;        /* size of wide-character buffer    */
```

The first parameter, uCodePage, specifies the codepage to be used when performing the conversion. This discussion applies to the first parameter of WideCharToMultiByte() as well. The codepage can be any valid codepage number. It is a good idea to check this number with IsValidCodepage(), even though MultiByteToWideChar() returns an error if an invalid codepage is used. The codepage may also be one of the following values:

```
CP_ACP       ANSI codepage
CP_OEMCP     OEM (original equipment manufacturer) codepage
```

CP_ACP instructs the API to use the currently set default Windows ANSI codepage. CP_OEMCP instructs the API to use the currently set default OEM codepage.

If Win32 ANSI APIs are used to get filenames from a Windows NT system, use CP_ACP when converting the string. Windows NT retrieves the name from the physical device and translates the OEM name into Unicode. The Unicode name is translated into ANSI if an ANSI API is called, then it can be translated back into Unicode with MultiByteToWideChar().

If filenames are being retrieved from a file that is OEM encoded, use CP_OEMCP instead.

MORE INFORMATION
================

When an application calls an ANSI function, the FAT/HPFS file systems will call AnsiToOem(); however, if an ANSI character does not exist in an OEM

codepage, the filename will not be representable. In these cases, SetFileApisToOEM() should be called to prevent this problem by setting a group of the Win32 APIs to use the OEM codepage instead of the ANSI codepage.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: WIntlDev

Multicolumn List Boxes in Microsoft Windows

PSS ID Number: Q64504

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In the Microsoft Windows environment, a multicolumn list box is designed to contain homogeneous data. For example, all the data might be "first names." These first names could logically fall into the same column or be in multiple columns. This feature was added to Windows at version 3.0 to enable a list box to be shorter vertically by splitting the data into two or three columns.

MORE INFORMATION

To create a multicolumn list box, specify the LBS_MULTICOLUMN style when creating the list box. Then the application calls the SendMessage function to send an LB_SETCOLUMNWIDTH message to the list box to set the column width.

When an application sends an LB_SETCOLUMNWIDTH message to a multicolumn list box, Windows does not update the horizontal scroll bar until the a string is added to or deleted from the list box. An application can work around this situation by performing the following six steps when the column width changes:

1. Send the LB_SETCOLUMNWIDTH message to the list box.

2. Send a WM_SETREDRAW message to the list box to turn off redraw.

3. Add a string to the list box.

4. Delete the string from the list box.

5. Send a WM_SETREDRAW message to the list box to turn on redraw.

6. Call the InvalidateRect function to invalidate the list box.

In response, Windows paints the list box and updates the scroll bar.

Windows automatically manages the list box, including horizontal and vertical scrolling and distributing the entries into columns. The distribution is dependent on the dimensions of the list box. Windows

fills Column 1 first, then Column 2, and so on. For example, if an application has a list box containing 13 ordered items and vertical space for 5 items, items 1-5 would be in the first column, items 5-10 in the second, and 11-13 in the last column, and item order would be maintained.

A multicolumn list box cannot have variable column widths.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Multiline Edit Control Does Not Show First Line

PSS ID Number: Q66668

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

When a multiline edit control is created that is less than one system character in height, the text in the edit control will not be displayed and subsequent attempts to enter text will cause the edit control to beep. This functionality is an invalid multiline edit control under Microsoft Windows versions 3.0 and later, even though this construct does work in Windows versions 2.x.

The multiline edit control also checks to see if the next line of text is displayable. If the next line of text is not displayable, it will beep to let you know that you have reached the limit of the edit control.

There is a similar situation with a control that overlaps another control in a dialog box. This construct is also considered invalid; thus, the second control will not be displayed.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Multiline Edit Control Limits in Windows NT

PSS ID Number: Q89712

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The default maximum size for a multiline edit (MLE) control in both Windows and Windows NT is 30,000 characters. The EM_LIMITTEXT message allows an application to increase this value. Setting "cchmax" to 0 is a portable method of increasing this limit to the maximum in both Windows and Windows NT. When cchmax is set to 0, the maximum size for an MLE is 4GB-1 (4 gigabytes minus 1).

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Multiline Edit Control Wraps Text Different than DrawText

PSS ID Number: Q67722

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Multiline edit controls will not wrap text in the same manner as the DrawText() function. This can be a problem when an application displays text that has been in an edit control because the text may wrap in a different location.

It is possible to obtain the text from the edit control and display it statically in a window with the same line breaks. To do this, the application must retrieve each line of text separately. This can be accomplished by sending the EM_GETLINE message to the control and displaying the retrieved text with the TextOut() function.

MORE INFORMATION

================

The following is a brief code fragment that demonstrates how to obtain the text of a multiline edit control line by line:

```
   ... /* other code */

   char  buf[80];         // Buffer for line storage
   HDC   hDC;             // Temporary display context
   HFONT hFont;           // Temporary font storage
   int   iNumEditLines;   // How much text
   TEXTMETRIC tm;         // Text metrics

   // Get number of lines in the edit control
   iNumEditLines = SendMessage(hEditCtl, EM_GETLINECOUNT, 0, 0L);

   hDC = GetDC(hWnd);

   // Get font currently selected into the control
   hFont = SendMessage(hEditCtl, WM_GETFONT, 0, 0L);

   // If it is not the system font, then select it into DC
   if (hFont)
      SelectObject(hDC, hFont);

   GetTextMetrics(hDC, &tm);
```

```
   iLine = 0;

   while (iNumEditLines--)
      {
      // First word of buffer contains max number of characters
      // to be copied
      buf[0] = 80;

      // Get the current line of text
      nCount = SendMessage(hEditCtl, EM_GETLINE, iLine, (LONG)buf);
      TextOut(hDC, x, y, buf, nCount);  // Output text to device
      y += tm.tmHeight;
      iLine++;
      }

   ReleaseDC(hWnd, hDC);
   ... /* other code */
```

The execution time of this code could be reduced by using the ExtTextOut() function instead of TextOut().

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Multimedia API Parameter Changes in the Win32 API

PSS ID Number: Q125864

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

SUMMARY

=======

This article discusses the following topics concerning multimedia APIs in the Win32 SDK:

- Some Windows version 3.1 APIs that require a device ID will also accept a properly cast device handle in the Win32 API if the caller is a 32-bit application.
- An explanation of the difference between using a device ID and using a device handle in conjunction with the above functions is given.
- A list of related APIs that are now obsolete but are provided for backwards compatibility with Windows version 3.1 is given.

MORE INFORMATION

================

Functions That Accept A Device Handle or Device ID

Several MIDI and wave audio APIs have been revised in Win32 to provide greater flexibility to application developers. The following APIs require the calling application to provide a device ID for the target device under Windows 3.1, but under Win32 the APIs also accept a properly cast device handle if the caller is a 32-bit application:

```
midiInGetDevCaps
midiOutGetDevCaps
waveInGetDevCaps
waveOutGetDevCaps
midiOutGetVolume
midiOutSetVolume
waveOutGetVolume
waveOutSetVolume
```

Device ID vs. Device Handle

A device ID has a one-to-one correspondence with the physical device it references and is determined by querying for the number of devices of a given type in the system and selecting the desired device. A device handle refers to a specific instance of a device, of which there may be more than one, and is obtained by opening a

device. A device instance may be thought of as a logical copy of a physical device.

If a 32-bit application is querying a device's capabilities using one of the xxxGetDevCaps APIs listed above, the distinction between whether a device ID or device handle is used in the function call is unimportant because all instances of a device have the same capabilities. The result of one of these function calls will be the same whether or not a device ID or device handle was used.

However, if a 32-bit application uses one of the xxxVolume APIs listed above to get or set the output volume of a device, the distinction between using a device ID or device handle becomes important. If a device ID is used in a call to these xxxVolume APIs, then the result of the call and/or information returned applies to all instances of the device. If a device handle is used in a call to the xxxVolume APIs, then the result of the call and/or information returned by the call applies only to the instance of the device referenced by the device handle.

The revised versions of the above APIs are available to 32-bit applications only. For backwards compatibility reasons, 16-bit applications are subject to the API design of Windows 3.1.

Obsolete APIs

In addition to the above changes, the following related APIs are now obsolete, but are included in Win32 for backwards compatibility purposes:

- midiInGetID
- midiOutGetID
- waveInGetID
- waveOutGetID

For further information about all the above APIs and how to use device IDs and device handles, please consult the Win32 Multimedia Programmer's Reference.

Additional reference words: 4.00 95
KBCategory: kbmm
KBSubcategory: MmMisc

Multimedia Group System and API Design Guidelines

PSS ID Number: Q67692

The information in this article applies to:

- Microsoft Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

The Microsoft Multimedia Systems Group is doing a large amount of system design and implementation. This article comments on the areas of system design.

MORE INFORMATION

Definitions

Term	Definition
Module	A module provides a set of functions and the interface to access those functions. The interface is called the API.
Client	A client uses a module. A client might be an application or a dynamic-link library (DLL).
Prefix	The initialization portion of the module that must be called before any of the other functions can be accessed.
Postfix	The closing portion of the module that must be called after the client has finished using the functions of the module.
Channel	A channel is created by calling a module's prefix function. It is used by the rest of the functions of the module (including the postfix function). Two notes about a channel: 1. A module can allocate resources to a channel. For example, a handle to a window is a channel, which has memory, a window procedure, and many other resources associated with it. 2. The channel uniquely identifies the user of the interface to the module. This allows the module to perform functions uniquely on each open channel. For example, each file handle represents a different file

that has been opened. Each file handle may have different attributes (read, write, read/write, binary, and so forth).

Separate Driver Interface from Module Interface

Separating the application from the hardware is one of the major tasks of system software. Layers within the operating system are separated from the hardware through the use of drivers. The system interface to the application should hide the mechanisms of the drivers as much as possible. This allows changing the mechanics of a driver in a later version of the system software. This also preserves the formality of the interface; applications are prevented from directly accessing the hardware.

Each application should also be separated from the drivers within the system. The driver API should not depend on where the driver is located, what it is named, or what form it takes (DLL, VxD, and so forth).

A driver should shield the application writer from its internal workings to enforce the principle of information hiding. If too much information is available, the application writer may choose to directly access the hardware, which jeopardizes the separation of functionality provided by the driver mechanism and system API.

Every Module Should Have an Initialization Function

Requiring the application to call a prefix function before using a system module and a postfix function after using a system module provides three major benefits, as follows:

1. Later versions of the driver can easily virtualize resources because the driver defies a channel for all communication. The driver can maintain separate resources for each communication channel, which is required for virtualization.

   Note that the complete virtualization does not need to be done in the first implementation of the module. As long as the interface requires the prefix and postfix calls, subsequent prefix calls can fail with an appropriate error message. Future versions of the system can properly handle multiple requests for the system resource.

2. The system module can allocate resources when the channel is allocated (when the prefix function is called), rather than when the system is initialized. Modules that are not called consume no resources. Resource allocation for all modules is postponed as long as possible.

3. The system can resolve conflicts between clients through the identification provided by the channels.

Interface Naming

During the process of designing a new module, use the interface naming conventions from an existing module with similar functionality. For example, an interface that deals with files would have a prefix interface containing the word "open," and a postfix interface containing the word "close." Extend naming conventions to similar functional areas. A stream interface would also have an "open" and a "close" function.

If the module has new functionality, then the functions can have unique names, such as the MIDI driver's midiOutStop function. If the module is similar to but not the same as another module, then use the function name to distinguish between modules. For example, the CreateWindow and CreateWindowEx functions in Windows create windows but CreateWindowEx also allows an application to specify other attributes.

The goal is to provide programmers familiar with existing modules a basis by which to quickly learn the new interface.

Prefix function names with the module name (abbreviated, if necessary). This allows the documentation to sort function names alphabetically, while keeping related functions together. More importantly, it allows easy identification of the module to which a function belongs.

Definition Naming

Prefix the names of constants and data structures with the module name (abbreviated, if necessary). For example, STRM_SEEK is a constant in the STRM group. This allows easy identification of the module to which a definition or data structure belongs.

As another example, MIDIOUTCAPS is the Device Capabilities structure for the MIDI output module. Using the naming convention developed here and symmetry (discussed in depth in the second part of this article), the MIDI Input module Device Capabilities structure should be called MIDIINCAPS.

Use additional prefixes as appropriate to identify the use of the definition. For example, MOERR_NODRIVER is a definition in the MIDI Output module to describe an error, that of no driver present.

Registering Drivers with Module

Most of the systems designed by the Multimedia group allow device drivers to be installed by the original equipment manufacturer (OEM) or even by the end user (given an appropriate setup program). There are two main ways for these drivers to communicate with the main module.

The first is to place an entry in the SYSTEM.INI file. When the parent module loads, it loads the child driver and initiates communication with the child.

The other method is for the child driver to call the parent to register itself as a client. This second method presumes that there is a suitable method available to load the child. Windows provides such a mechanism.

Requiring a driver to register itself with the handler module provides four benefits:

1. Drivers can be installed by adding them to the "modules to load" list. This is much easier than creating a line for the SYSTEM.INI file.

2. The handler module is more general because it does not assume the presence of certain drivers. This enhances system portability and reduces interdependencies between drivers and handlers. This advantage also applies to drivers loaded by a parent process.

3. A driver can pass information about itself, such as its name and entry points, to its parent during registration. This further separates the parent module from the driver. As long as the format of the interface data is fixed, independent changes may be made to both parent and driver.

4. Run-time installation of drivers is possible. The inherent nature of registration makes installing new drivers while the system is running much easier. This also simplifies implementing virtualization.

Symmetry of Function Names

- Every Open function should have a Close function and every Get function a Put function.

- Related functions in separate areas should work the same way. For example, if the MIDI output has an Open, the MIDI input should also have an Open. Additionally, the return values and parameters should be as similar as possible. This eases the programmer's task of learning the new APIs. This applies even if the current implementation doesn't use API symmetry. See "Designing for Implementation in Steps," below.

Symmetry in Naming Conventions

- Name defined constants and types for related areas should all be named using the same conventions. For example, LPMIDICALLBACK and LPWAVECALLBACK.

- If a naming convention already exists for a function type, adhere to it. Example: use SEEK and TELL functions to move within a file

system.

- If any part of an existing convention is used, little deviation from it is allowed. For example, a combination of SEEK and GET functions to move within a file system would not be the product of good design because it confuses an existing convention.

- If a convention does not already exist, create a new naming convention to avoid confusing things. Example: KNOCK and ANSWER.

Design for Implementation in Steps

Most implementations of any size must be done in incremental steps of functionality. More and more features are added to the modules until the entire design is completely implemented. For large or complex modules, this process may occur over several years. However, the original design must anticipate the complete, final functionality, not just the short-term goals. For example, even if allowing multiple users of a module will not be implemented in the first phase, this capability should be designed into the API. That way, the impact on users of the module will be minimal once implementation is complete.

Avoid placing arbitrary limits on functionality due to details of the current implementation. For example, even if only one user can have a resource allocated today, this may not always be true. Specifically, the Open function should return a handle to the resource that is then passed to functions that manipulate the resource. In the future, when multiple users of the resource is implemented, it will not be necessary to change other functions or applications.

In a message-based system, functions should return a "message not recognized" code for unexpected messages that is distinct from the "an error occurred" code. Then, when a future version of the driver contains extended functionality, an application can determine if the installed version of the driver supports the new features. If not, the application can take appropriate alternative action.

A project designed to be built in phases has well defined progress milestones. This makes it much easier to track progress while the module is under construction.

Building a module in phases also makes it easier to verify that the module is built correctly. Testing receives increments of functionality instead of the entire product toward the end of the development cycle.

Error Reporting

An function call can fail for many reasons. It is best if the call can return the specific cause of the error in addition to noting that the call failed. Functions that return a handle, structure, or other data cause particular problems because there is a limited set of values that are always invalid.

Three approaches to error reporting are:

1. Ignore it (not recommended).

2. Provide a separate "what was that error?" call. This is more complicated than it sounds because, in a multitasking system, there can be multiple users of the module at the same "time." This makes determining what was the last error for a particular application difficult.

3. Return the handle or structure in a parameter and return the error code as the function return. This seems to be the best option, and is the approach used by OS/2.

Now that the error code is available, what should be done with it? To allow for internationalization and for additional error codes, the application should not associate the error code with a message. Instead, provide a function in each API that returns the text message for a specified error code. This function might be named GetTextErrorInformation, for example.

Client-Supplied Buffers

It is desirable for the client application to provide all buffers that it will access. If a system module allocates and maintains buffers, many implementation problems can arise when a buffer is made visible to the client application. Three advantages of client-supplied buffers are:

1. If the system software runs at a different privilege level or on a different CPU, or is otherwise separate from the client application, the system software can easily access the buffer. However, at the client's lower privilege level, or if the client and operating system are on different CPUs, it may be extremely difficult (if not impossible) to make a system-supplied buffer available to the client.
2. When the application supplies the buffers, the application has complete control over how much memory the system module uses.

3. The application is responsible for reporting an out-of-memory error. This removes an error condition from the system call.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbmm kbprg
KBSubcategory: MMMisc

Multiple Columns of Text in Windows Help Files

PSS ID Number: Q67895

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

Microsoft Windows Help version 3.0 will support multiple columns of information in Help files. Unfortunately, the Table features of Word for Windows are not supported in Help version 3.0. In Windows Help version 3.1, however, Word for Windows tables are supported. Below is an outline of the supported techniques for creating multiple columns in Help files.

MORE INFORMATION

The best results can be achieved with Help version 3.0 if the text is organized such that one column has the bulk of the text to be presented and the other columns have relatively little text. The text might resemble the following:

```
   column1column1  column2  column3column3column3column3column3column3col
                            column3column3column3...
```

As the size of the Help window is decreased, the text in the third column will wrap to remain displayed for as long as possible. To achieve this effect, format the text as paragraphs with an "outdent," or negative indent, as shown in the following example:

```
   First line              Left margin and tab stop
   v                       v
   column1column1  column2  column3column3column3column3column3column3col
                            column3column3column3...
```

A typical format for paragraphs of this type is:

```
   Left Margin 2.5"
   First Line -2.5"
```

It is also necessary to define a tab stop at 2.5 inches.

Having two columns of text in the outdent is merely an example. You can define as many or as few columns in that space as necessary.

If two columns with similar amounts of text in each are required, you can use the side-by-side paragraph formatting of Word for the Macintosh or Word for MS-DOS. Only two paragraphs side-by-side are supported by the Help compiler.

If neither of the tools mentioned above is available, or if more complicated tables are required, you can format the tables manually. Define appropriate tab stops and use them to align the columns of text. Format each physical line in the table as a separate paragraph. Select the entire table and format the paragraphs as "Keep Together." In the context of a Help system, this paragraph format disables word wrap.

The following is an example of a more complex table. In this example, (P*) is the paragraph mark at the end of a line:

```
   column1column1   column2column2  column3column3column3column3colu(P*)
   column1column1   column2coul     column3column3column3column3colu(P*)
   column1column1                   column3column3column3column3(P*)
   column1column1(P*)

   newcol1newcol1   newcol2newcol2  newcol3newcol3newcol3newcol3...(P*)
```

As the size of the Help window is reduced, the text will not wrap. Instead, you must use the horizontal scroll bar at the bottom of the window to view the remainder of the table.

Support for Word for Windows tables was added in Windows Help version 3.1.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsHlp

Multiple References to the Same Resource

PSS ID Number: Q83808

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Windows supports multiple references to a given resource. For example, suppose that an application has two top-level menus that each contain the same submenu. (An application can use the AppendMenu or SetMenu functions to add a submenu to another menu at run time.)

Normally, destroying a menu destroys all of its submenus. In the case above, however, when one menu is destroyed, the other menu has a lock on the common submenu. Therefore, the common submenu remains in memory and is not destroyed. The handle to the submenu remains valid until all references to the submenu are removed. The submenu either remains in memory or is discarded, while its handle remains valid.

MORE INFORMATION
================

Windows maintains a lock count for each resource, including menus. When the lock count falls to zero, Windows can free (destroy) the object. Each time an application loads a resource, its lock count is incremented. If a resource is loaded more than once, only one copy is created; subsequent loads only increment the lock count. Each call to free a resource decrements its lock count.

When the LoadResource function determines if a resource has already been loaded, it also determines if the resource has been discarded. If so, LoadResource loads the resource again. The resource is not necessarily present in memory at all times. However, if the lock count is not zero and the resource is discarded, Windows will automatically reload the resource. All resources are discardable and will be discarded if required to free memory.

Therefore, in the example above, the application's call to the DestroyMenu function calls FreeResource, which checks the lock count. This process is analogous to LoadMenu, which calls LoadResource.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrRsc

Multiprotocol Support for Windows Sockets

PSS ID Number: Q125704

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

SUMMARY

=======

The Windows Sockets (WinSock) API is based on the Berkeley Sockets progamming model, the standard interface for TCP/IP network programming on Windows. However, the WinSock implementations for Windows NT and Windows 95 include support for additional network transports. Here are the network transports supported by Microsoft WinSock implementations, listed by platform. For convenience, the headers and libraries required for application development are also listed.

Platform	Transport	Header*	Lib
Windows NT	TCP/IP	WINSOCK.H	WSOCK32.LIB
	IPX/SPX	WSIPX.H, WSNWLINK.H	""
	NetBEUI (via NetBIOS)	WSNETBS.H	""
	Appletalk	ATALKWSH.H	""
	ISO/TP4	WSHISOTP.H	""
Windows 95	TCP/IP	WINSOCK.H	""
	IPX/SPX	WSIPX.H, WSNWLINK.H	""
Windows for Workgroups version 3.11	TCP/IP	WINSOCK.H	WINSOCK.LIB

* WINSOCK.H is required for all platforms and transports, in addition to other header files.

MORE INFORMATION

================

The Windows Sockets API provides a uniform interface to multiple network transports and shields the programmer from most transport level ideosyncracies. However, WinSock does not eliminate the need to learn the basics of the transport protocol used. In particular, the programmer should be familiar with the following aspects of any transport protocol used with Windows Sockets:

1. Addressing.

   Each transport uses different address format. For example, the IP socket address structures looks like this:

```
   /*
    * Socket address, internet style.
```

```
 */
struct sockaddr_in
{
   short   sin_family;
   u_short sin_port;
   struct  in_addr sin_addr;
   char    sin_zero[8];
};

struct in_addr
{
   union
   {
      struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
      struct { u_short s_w1,s_w2; } S_un_w;
      u_long S_addr;
   } S_un;
}
```

In contrast, the IPX address structure looks like this:

```
typedef struct sockaddr_ipx
{
   short sa_family;
   char  sa_netnum[4];
   char  sa_nodenum[6];
   unsigned short sa_socket;
} SOCKADDR_IPX;
```

2. Connection-Oriented vs. Connectionless Transport.

   In a connection-oriented transport protocol such as TCP or SPX, applications are required to establish a virtual circuit before data transfer can take place. The following sequences of WinSock functions are required at minimum to establish a virtual circuit:

```
Server     Client
------------------
socket     socket
bind       connect
listen
accept
```

   After the virtual circuit is established, the send() and recv() functions are used to transfer data.

   In a connectionless transport, a virtual circuit is not established. Both the client and server exchange data by binding a socket and calling sendto() or recvfrom().

3. Virtual circuit termination semantics.

4. Message Oriented vs. Stream-Oriented.

   See the Win32 SDK online documentation for information on these topics.

5. Expedited data delivery.

   This is data that has been earmarked by the application as urgent data, and will be sent by the transport as quickly as possible. Not all transports support this feature. The Windows Sockets API provides the ability to request expedited data delivery via the MSG_OOB flag in the send() function.

6. Broadcasts.

   Here is an extract from the Win32 SDK online documentation:

   "Most connectionless transport protocols support broadcasts in the same fashion, in which any bound socket can send a broadcast if the SO_BROADCAST option is set, and broadcasts sent to the appropriate local endpoint are received without any additional work on the part of the application. NetBIOS transport protocols, however, handle broadcasts somewhat differently. In order to receive broadcasts, an application must bind to the NetBIOS broadcast address, which is an asterisk ("*") followed by 15 blank spaces (ASCII character 0x20). This means two things: A socket must be specially bound to receive broadcasts, and applications cannot depend on receiving broadcasts intended only for a specific application, since all NetBIOS broadcasts are delivered to this address. In other protocols such as UDP/IP and IPX, broadcasts are delivered to a socket only if the broadcast was sent to the same port to which the socket was bound."

For more information on any of the above topics, please see the Win32 SDK online documentation.

Additional reference words: 4.00
KBCategory: kbnetwork
KBSubcategory: NtwkWinsock

Mutex Wait Is FIFO But Can Be Interrupted

PSS ID Number: Q125657

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

A mutex is a synchronization object that is signalled when it is not owned by any thread. Only one thread at a time can own a mutex. Other threads requesting the mutex will have to wait until the mutex is signalled. This article discusses the order in which threads acquire the mutex.

MORE INFORMATION
================

Threads that are blocked on a mutex are handled in a first in, first out (FIFO) order. Therefore, the first to wait on the mutex will be the first to receive the mutex, regardless of thread priority.

It is important to remember that Windows NT can interrupt the wait and that this will change the order in which threads are queued. A kernel-mode asynchronous procedure call (APC) can interrupt a user-mode thread's execution at any time. Once the normal execution of the thread resumes, the thread will again wait on the mutex; however, the thread is placed at the end of the wait queue. For example, each time you enter the debugger (hit a breakpoint, execute OutputDebugString(), and so on), all application threads are suspended. Suspending a thread causes the thread to run a piece of code in kernel mode. When you continue from the debugger, the threads are resumed, causing them to resume their wait for the mutex, but possibly in a different order than before. In this case, it does not look like the mutex is acquired in FIFO order. Some threads may be unable to acquire the mutex when the application in run under the debugger.

NOTE: This implementation detail is subject to change. Windows 95 and other platforms that support the Win32 API may adopt different strategies.

Most programs do not usually need to make any assumption about this behavior. The only class of applications that should be sensitive to mutex claim and release throughput are realtime-class applications. If throughput is of importance to a program, critical sections should be used wherever possible.

Additional reference words: 3.10 3.50 windbg ntsd
KBCategory: kbprg
KBSubcategory: BseSync

Named Pipe Buffer Size

PSS ID Number: Q105531

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

The documentation for CreateNamedPipe() indicates that

> The input and output buffer sizes are advisory. The actual buffer size reserved for each end of the named pipe is either the system default, the system minimum or maximum, or the specified size rounded up to the next allocation boundary.

The buffer size specified should be a reasonable size so that your process will not run out of nonpaged pool, but it should also be large enough to accommodate typical requests.

Every time a named pipe is created, the system creates the inbound and/or outbound buffers using nonpaged pool, which is the physical memory used by the kernel. The number of pipe instances (as well as objects such as threads and processes) that you can create is limited by the available nonpaged pool. Each read or write request requires space in the buffer for the read or write data, plus additional space for the internal data structures.

Whenever a pipe write operation occurs, the system first tries to charge the memory against the pipe write quota. If the remaining pipe write quota is enough to fulfill the request, the write completes immediately.

If the remaining pipe write quota is too small to fulfill the request, the system will try to expand the buffers to accommodate the data using nonpaged pool reserved for the process. The write will block until the data is read from the pipe so that the additional buffer quota can be released. Therefore, if your specified buffer size is too small, the system will grow the buffer as needed, but the downside is that the operation will block. If the operation is overlapped, a system thread is blocked; otherwise, the application thread is blocked.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseIpc

Named Pipes Under WOW

PSS ID Number: Q94948

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

A 16-bit Windows-based application under Windows on Win32 (WOW) may use a named pipe that was created previously by a Win32-based application; however, REDIR.EXE must be included in the AUTOEXEC.BAT file for this to work properly.

Note that for local named pipes, networking doesn't need to be enabled.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: SubSys

Nesting Quotation Marks Inside Windows Help Macros

PSS ID Number: Q77748

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

In Windows Help macros, strings may be delimited in two ways. The string can be opened and closed by double quotation marks or the string can be opened by a single opening quotation mark and closed by a single closing quotation mark.

Any quoted strings contained in a string delimited with double quotation marks must be enclosed in opening and closing single quotation marks.

The single opening quotation mark is different from the single closing quotation mark. The single opening quotation mark (`) is paired with the tilde (~) above the TAB key on extended keyboards; the single closing quotation mark (') is the same as the apostrophe. For example,

```
   CreateButton("time_btn", "&Time", "ExecProgram("clock", 0)")
```

is illegal because the string "clock" uses double quotation marks within the double quotation marks used for the ExecProgram macro. The following example corrects the error by enclosing "clock" in single quotation marks:

```
   CreateButton("time_btn", "&Time", "ExecProgram(`clock', 0)")
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsHlp

NetBIOS Name Table and NCBRESET

PSS ID Number: Q95944

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY

The Windows NT NetBIOS implementation conforms to the IBM NetBIOS 3.0 specifications, with several enhancements discussed in this article.

MORE INFORMATION

Name Table

Under Windows NT, the name table is maintained on a per-process basis, which means that names added by one process are not visible by a different process. This also means that for two processes to establish a session, both processes must register two different NetBIOS names. However, sessions can be established by two threads in the same process using the same NetBIOS name.

NCBRESET

The IBM NetBIOS 3.0 specifications defines four basic NetBIOS environments under the NCBRESET command. Win32 follows the OS/2 Dynamic Link Routine (DLR) environment. This means that the first NCB issued by an application must be a NCBRESET, with the exception of NCBENUM. The Windows NT implementation differs from the IBM NetBIOS 3.0 specifications in the NCB_CALLNAME field.

In the "IBM Local Area Network Technical Reference," under the section on NetBIOS 3.0, the NCB_CALLNAME field is defined as the following:

   REQ_SESSIONS at NCB_CALLNAME+0 (1-byte field)
   The number of sessions requested by the application program.
   If zero, the default of 16 is used.

   REQ_COMMANDS at NCB_CALLNAME+1 (1-byte field)
   The number of commands requested by the application program.
   If zero, the default of 16 is used.

   REQ_NAMES at NCB_CALLNAME+2 (1-byte field)
   The number of names requested by the application program. This
   does not include a reservation for NAME_NUMBER_1.
   If zero, the default of 8 is used.

REQ_NAME_ONE at NCB_CALLNAME+3 (1-byte field)
A request to reserve NAME_NUMBER_1 for this application program.
If 0, NAME_NUMBER_1 is not requested.
If not 0, NAME_NUMBER_1 is desired to be reserved for this application.

Under the Windows NT implementation, the REQ_COMMANDS (NCB_CALLNAME+1) field is ignored. Instead, an application is bound by the amount of memory the process can allocate.

For more information on the differences between the Windows NT implementation and the IBM NetBIOS 3.0 specifications, see "The NetBIOS Function" in the "Win32 API Reference" Help file.

For more information on version 3.0 of NetBIOS, contact IBM and order the "IBM Local Area Network Technical Reference."

Additional reference words: 3.00 3.10 3.50 NCBRESET
KBCategory: kbnetwork
KBSubcategory: NtwkNetbios

Network DDE For 16-bit Windows-based Apps Under Windows NT

PSS ID Number: Q127861

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.5

Network Dynamic Data Exchange (NetDDE) has limited support for 16-bit Windows application running under Windows NT. You can use DDE across the network, however, the NDde APIs are not supported.

The NDde APIs, such as NDdeShareAdd(), are used to create and manage the NetDDE shares, not for the actual communication. Therefore, for 16-bit applications to use NetDDE under Windows NT, you will need to use Generic Thunks to thunk to the 32-bit NDde APIs to create and trust the share. Once that is done, you can communicate using DDE or DDEML.

NOTE: You must be an administrator to add a DDE share.

Additional reference words: 3.50
KBCategory: kbui
KBSubcategory: SubSys UsrNetDde

New Dialog Styles in Windows 95

PSS ID Number: Q125678

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SUMMARY

Windows 95 provides a few new dialog styles -- all listed in this article. All Windows version 3.1 dialog styles are still usable in Windows 95. However, DS_LOCALEDIT cannot be used in Win32-based applications because it does not apply. Although not documented, DS_ABSALIGN and DS_SETFONT exist in Windows version 3.1. They are documented in Windows 95.

MORE INFORMATION

Here is a list of the new dialog styles:

DS_3DLOOK	Gives the dialog box a nonbold font and draws three-dimensional borders around control windows in the dialog box.
DS_CENTER	Centers the dialog box in the working area -- the area not obscured by the tray.
DS_CENTERMOUSE	Centers the mouse cursor in the dialog box.
DS_CONTEXTHELP	Includes a question mark in the title bar of the dialog box. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a control in the dialog box, the control receives a WM_HELP message. The control should pass the message to the dialog procedure, which should call the WinHelp function using the HELP_WM_HELP command. The Help application displays a pop-up window that typically contains help for the control. Note that DS_CONTEXTHELP is just a placeholder. When the dialog box is created, the system checks for DS_CONTEXTHELP and, if it is there, adds WS_EX_CONTEXTHELP to the extended style of the dialog box.
DS_CONTROL	Creates a dialog box that works well as a child window of another dialog box, much like a page in a property sheet. This style allows the user to tab among the control windows of a child dialog box, use its accelerator keys, and so on.

DS_FIXEDSYS	Uses SYSTEM_FIXED_FONT instead of SYSTEM_FONT.
DS_NOFAILCREATE	Creates the dialog even if errors occur -- for example, if a child window cannot be created or if the system cannot create a special data segment for an edit control.
DS_SETFOREGROUND	Brings the dialog box to the foreground. Internally, Windows calls the SetForegroundWindow function for the dialog box.

Additional reference words: 4.00 DLGTEMPLATE kbinf
KBCategory: kbui
KBSubcategory: UsrDlgs

New Owner in Take-Ownership Operation

PSS ID Number: Q111541

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

When ownership of a file is taken, the user performing the operation becomes the new owner. The exception to this rule is when the user is a member of the "Administrators" group. In this situation, the ownership of the file is assigned to the Administrators group.

The reasoning for this behavior is that the administrators on a particular system work together. When one administrator takes ownership of a file, the others should also receive access.

MORE INFORMATION

================

When a take-ownership operation is performed, the system assigns the new owner SID based on the TOKEN_OWNER field of the user's access token.

When a user logs on to a Windows NT system, the logon process builds an access token to represent the user. Normally the TOKEN_OWNER field in the access token is set equal to TOKEN_USER (the user's SID). However, when the user is a member of the Administrators group, the system sets the TOKEN_OWNER field to the Administrators SID.

Although Windows NT does not provide a user interface for changing the TOKEN_OWNER field in the user's access token, it is possible to programatically change this value via the SetTokenInformation() Win32 API (application programming interface).

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

New User Heap Limits Under Windows 95

PSS ID Number: Q125676

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

In Windows version 3.1, window and menu data is maintained in two 16-bit heaps. This limits window and menu data to 64k each. Windows 95 uses 32-bit heaps for window and menu data, thus greatly expanding the limits placed on the number of items contained in these heaps.

MORE INFORMATION
================

Windows version 3.1 user and menu heaps are each limited to 64K of data. As a result, the number of window and menus in a system are each constrained to around 200. In Windows 95, the number of Windows and Menus that may exist in the system goes up to 32K each. This is possible because the Windows 95 user and menu heaps are each two megabytes in size.

The first 64K of the user heap looks exactly as it did in Windows version 3.1, except for the absense of WND structures. The Windows 95 WND structures populate the heap space above 64K, thus increasing the number of WND structures the heap can hold. This new arrangement also has the positive effect of freeing up space in the lower 64K space, making more space for class structures and other items that reside in the user heap. For the menu heap, there is nothing special about the low 64K, menus and their data may appear anywhere in the two-megabyte heap.

Additional reference words: 4.00
KBCategory: kbusage
KBSubcategory: UsrMisc

New Window Styles in Windows 95

PSS ID Number: Q125679

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY

Windows 95 provides a number of new window styles that help make the user interface more attractive and intuitive. These new styles are listed in this article. Most of the new styles are extended styles, which are specified with the CreateWindowsEx() function. All Windows version 3.1 window styles are still usable in Windows 95.

MORE INFORMATION

Here are the new styles:

WS_EX_ABSPOSITION	Specifies that a window has an absolute position.
WS_EX_CLIENTEDGE	Specifies that a window has a 3D look -- that is a border with a sunken edge.
WS_EX_CONTEXTHELP	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message.
WS_EX_CONTROLPARENT	Allows the user to navigate among the child windows of the window by using the TAB key.
WS_EX_LEFT	Gives window generic left-aligned properties. This is the default.
WS_EX_LEFTSCROLLBAR	Places vertical scroll bar to the left of the client area.
WS_EX_LTRREADING	Displays the window text using left-to-right reading order properties. This is the default.
WS_EX_MDICHILD	Creates an MDI child window.
WS_EX_OVERLAPPEDWINDOW	Combines the WS_EX_CLIENTEDGE and WS_EX_WINDOWEDGE styles.
WS_EX_PALETTEWINDOW	Combines the WS_EX_WINDOWEDGE, WS_EX_SMCAPTION, and WS_EX_TOPMOST styles.

WS_EX_RIGHT	Gives window generic right-aligned properties. This depends on the window class.
WS_EX_RIGHTSCROLLBAR	Places vertical scroll bar (if present) to the right of the client area. This is the default.
WS_EX_RTLREADING	Displays the window text using right-to-left reading order properties.
WS_EX_STATICEDGE	Creates a window with a three-dimensional border style intended to be used for items that do not accept user input.
WS_EX_TOOLWINDOW	Creates a tool window, which is a window intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the task bar or in the window that appears when the user presses ALT+TAB.
WS_EX_WINDOWEDGE	Specifies that a window has a border with a raised edge.

Additional reference words: 4.00 CreateWindowEx kbinf
KBCategory: kbui
KBSubcategory: UsrWndw

New Windows 95 Styles Make Attaching Bitmap to Button Easier

PSS ID Number: Q125673

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY

=======

In Windows 95, there are two new button styles (BS_BITMAP and BS_ICON). Using these styles makes attaching a bitmap or an icon to a button in Windows 95 easier than it was in Windows version 3.1.

MORE INFORMATION

================

In Windows version 3.1 you create the button by using the CreateWindow() function with the BS_OWNERDRAW style to attach a bitmap or an icon to a button. Each time the parent window receives the WM_DRAWITEM message the bitmap or the icon must be loaded and drawn on the button.

In Windows 95 you can create a button with style BS_BITMAP to display a bitmap instead of text on the button. After you create the button by using CreateWindow() function, assign the bitmap to the button by sending a WM_SETIMAGE message to the button with the wparam as IMAGE_BITMAP and lparam as a handle to the bitmap. Windows displays the specified bitmap on the button. The attached bitmap should not be deleted until the button uses it.

SAMPLE CODE

===========

```
hwndButton = CreateWindow(
               "BUTTON",   // predefined class
               "OK",       // button text
               WS_VISIBLE| WS_CHILD | BS_DEFPUSHBUTTON |BS_BITMAP, //styles
               // Size and position values are given explicitly, because
               // the CW_USEDEFAULT constant gives zero values for buttons.
               5, // starting x position
               5, // starting y position
               30,// button width
               18, // button  height
               hWnd,// parent window
               NULL,// No menu
               (HINSTANCE) GetWindowLong(hWnd,
                           GWL_HINSTANCE), NULL); // pointer not needed

// load the bitmap, NOTE: delete it only when it's no longer being used.
hBitmap   = LoadBitmap(hInst,MAKEINTRESOURCE(IDB_BITMAP1));
// associate the bitmap with the button.
```

```
SendMessage(hwndButton,BM_SETIMAGE,(WPARAM)IMAGE_BITMAP,
            (LPARAM)(HANDLE)hBitmap);
```

In Windows 95, you can also create a button with style BS_ICON to display a icon instead of text on the button. After creating the button by using CreateWindow() function, assign the icon to the button by sending a WM_SETIMAGE message to the button with the wparam as IMAGE_ICON and the lparam as the handle to the icon. Windows displays the specified icon on the button.

NOTE: The system handles cleanup of icons or cursors loaded from resources. Therefore, the icon should not be deleted unless the icon was created at run time by using CreateIcon() function.

SAMPLE CODE
==========

```
hwndButton = CreateWindow(
                  "BUTTON",   // predefined class
                  "OK",       // button text
                  WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON|BS_ICON,//styles
                // Size and position values are given explicitly, because
                // the CW_USEDEFAULT constant gives zero values for buttons.

                   5, // starting x position
                   5, // starting y position
                  30, // button width
                  18, // button height
                  hWnd,   // parent window
                  NULL,   // No menu
                  (HINSTANCE) GetWindowLong(hWnd,GWL_HINSTANCE),
                  NULL);      // pointer not needed

// load the icon, NOTE: you should delete it after assigning it.
hIcon = LoadIcon(hInst,MAKEINTRESOURCE(IDI_ICON1));
// associate the icon with the button.
SendMessage(hwndButton,BM_SETIMAGE,(WPARAM)IMAGE_ICON,
            (LPARAM)(HANDLE)hIcon);
```

If a bitmap or icon is not specified via the BM_SETIMAGE message and the BS_BITMAP or BS_ICON style is specified, a blank button with no text is displayed.

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrWnd

No Subsystem Developer Kit or Support for Such a Kit

PSS ID Number: Q89987

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

There is not a kit available for developers to write subsystems, nor are there plans to provide such a kit. Furthermore, there is no documentation or support available to develop a subsystem for Windows NT.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: SubSys

Non-Addressable Range in Address Space

PSS ID Number: Q92764

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

In Windows NT, each process has its own private address space. The process can use up to 2 gigabytes of virtual memory. This 2Gb is not necessarily contiguous. The system uses the other 2Gb.

The user-mode addresses extend from 0x00010000 to 0x7FFF0000. The following ranges are reserved as non-address space to ensure that the process does not walk on system-owned memory

   0x00000000 to 0x0000FFFF   (first 64K of virtual space)

   0x7FFF0000 to 0x7FFFFFFF   (last 64K of user virtual space)

These are effectively PAGE_NOACCESS ranges.

Additionally, Win32 DLLs will reserve other specific address ranges. For more information, see the file COFFBASE.TXT that comes with the DDK.

MORE INFORMATION

================

This range is not guaranteed to serve this purpose in the future. There could be good reasons in a future implementation to use these addresses. Code that is going to depend on this non-address range should verify its validity at run time with something like

```
BOOL IsFirst64kInvalid(void)
{
   BOOL bFirst64kInvalid = FALSE;

   try {
      *(char *)0x0000FFFF;
   }
   except (EXCEPTION_EXECUTE_HANDLER) {
      if (EXCEPTION_ACCESS_VIOLATION == GetExceptionCode())
         bFirst64kInvalid = TRUE;
   }

   return bFirst64kInvalid;
}
```

Additional reference words: 3.10 3.50

KBCategory: kbprg
KBSubcategory: BseMm

Noncontinuable Exceptions

PSS ID Number: Q98840

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

An exception is noncontinuable if the event isn't continuable in the hardware, or if continuation makes no sense. For example, if the caller's stack is corrupted while trying to post an exception, continuing from the bad stack exception would not be useful.

The noncontinuable exception does not terminate the application, and therefore an application that can succeed in catching the exception and running after a noncontinuable exception is free to do so. However, a noncontinuable exception typically arises as a result of a corrupted stack or other serious problem, making it very difficult to recover from the exception.

Additional reference words: 3.10 3.50 4.00 95 non-continuable
KBCategory: kbprg
KBSubcategory: BseExcept

Nonzero Return from SendMsg() with HWND_BROADCAST

PSS ID Number: Q102588

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The SendMessage() function calls the window procedure for the specified window and does not return until that window has processed the message and returned a value. Applications can send messages to all top-level windows in the system by specifying HWND_BROADCAST as the first parameter to the SendMessage() function. In doing so, however, applications lose access to the return values resulting from the SendMessage() call to each of the top-level windows.

MORE INFORMATION

When a call to SendMessage() is made, the value returned by the window procedure that processed the message is the same value returned from the SendMessage() call.

Among other things, SendMessage() determines whether the first parameter is HWND_BROADCAST (defined as -1 in WINDOWS.H). If HWND_BROADCAST is the first parameter, SendMessage enumerates all top-level windows in the system and sends the message to all these windows. Because this one call to SendMessage() internally translates to a number of SendMessage() calls to the top-level windows, and because SendMessage() can return only one value, Windows ignores the individual return values from each of the top-level window procedures, and just returns a nonzero value to the application that broadcast the message. Thus, applications that want to broadcast a message to all top-level windows, and at the same time expect a return value from each SendMessage() call, should not specify HWND_BROADCAST as the first parameter.

There are a couple of ways to access the correct return value from messages sent to more than one window at a time:

- If the broadcasted message is a user-defined message, and only a few other applications respond to this message, then those applications that trap the broadcasted message must return the result by sending back another message to the application that broadcast the message. The return value can be encoded into the message's lParam.

- If the application does not have control over which application(s) will respond to the message, and it still expects a return value, then the application must enumerate all the windows in the system using EnumWindows() function, and send the message separately to each window it obtained in the enumeration callback function.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMsg

Objects Inherited Through a CreateProcess Call

PSS ID Number: Q83298

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

The objects inherited by a process started by a call to CreateProcess() are those objects that you can get a handle to and on which you can use the CloseHandle() function. These objects include the following:

- Processes
- Events
- Semaphores
- Mutexes
- Files (including file mappings)
- Standard input, output, or error devices

However, the new process will only inherit objects that were marked inheritable by the old process.

These are duplicate handles. Each process maintains memory for its own handle table. If one of the processes modifies its handle (for example, closes it or changes the mode for the console handle), other processes will not be affected.

Processes will also inherit environment variables, the current directory, and priority class.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseProcThrd

Obsolete Macro functions in Japanese Windows Version 3.1

PSS ID Number: Q130059

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Win32s version 1.2

SUMMARY

Some of the Japanese Windows version 3.1 functions are no longer supported in the Japanese version of Windows NT. These Macro functions are obsolete in the current C++ compiler.

MORE INFORMATION

The macro functions of Japanese Windows version 3.10 are no longer supported and are obsolete in current C++ compiler. They were remapped to other generic MBCS functions. Use the following as workaround functions. Include them in the current header file. T _ismbb* entries correspond to the replaced Windows version 3.1 macro.

```
#include <mbctype.h>
#define iskana       _ismbbkana
#define iskpun       _ismbbkpunct
#define iskmoji     _ismbbkalpha
#define isalkana    _ismbbalpha
#define ispnkana    _ismbbpunct
#define isalnmkana  _ismbbalnum
#define isprkana    _ismbbprint
#define isgrkana    _ismbbgraph
#define iskanji     _ismbblead
#define iskanji2    _ismbbtrail
```

Additional reference words: 3.10 1.20 3.50 kbinf
KBCategory: kbother
KBSubcategory: wintldev

Obtaining a Console Window Handle (HWND)

PSS ID Number: Q124103

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

It may be useful to manipulate a window associated with a console application. The Win32 API provides no direct method for obtaining the window handle associated with a console application. However, you can obtain the window handle by calling FindWindow(). This function retrieves a window handle based on a class name or window name.

Call GetConsoleTitle() to determine the current console title. Then supply the current console title to FindWindow().

MORE INFORMATION
================

Because multiple windows may have the same title, you should change the current console window title to a unique title. This will help prevent the wrong window handle from being returned. Use SetConsoleTitle() to change the current console window title. Here is the process:

1. Call GetConsoleTitle() to save the current console window title.

2. Call SetConsoleTitle() to change the console title to a unique title.

3. Call Sleep(40) to ensure the window title was updated.

4. Call FindWindow(NULL, uniquetitle), to obtain the HWND
   this call returns the HWND -- or NULL if the operation failed.

5. Call SetConsoleTitle() with the value retrieved from step 1, to
   restore the original window title.

You should test the resulting HWND. For example, you can test to see if the returned HWND corresponds with the current process by calling GetWindowText() on the HWND and comparing the result with GetConsoleTitle().

The resulting HWND is not guaranteed to be suitable for all window handle operations.

Sample Code

The following function retrieves the current console application window handle (HWND). If the function succeeds, the return value is the handle of the console window. If the function fails, the return value is NULL. Some error checking is omitted, for brevity.

```
HWND GetConsoleHwnd(void)
{
    #define MY_BUFSIZE 1024 // buffer size for console window titles
    HWND hwndFound;         // this is what is returned to the caller
    char pszNewWindowTitle[MY_BUFSIZE]; // contains fabricated WindowTitle
    char pszOldWindowTitle[MY_BUFSIZE]; // contains original WindowTitle

    // fetch current window title

    GetConsoleTitle(pszOldWindowTitle, MY_BUFSIZE);

    // format a "unique" NewWindowTitle

    wsprintf(pszNewWindowTitle,"%d/%d",
                GetTickCount(),
                GetCurrentProcessId());

    // change current window title

    SetConsoleTitle(pszNewWindowTitle);

    // ensure window title has been updated

    Sleep(40);

    // look for NewWindowTitle

    hwndFound=FindWindow(NULL, pszNewWindowTitle);

    // restore original window title

    SetConsoleTitle(pszOldWindowTitle);

    return(hwndFound);
}
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbui kbcode
KBSubcategory: BseCon UsrWndw

Obtaining Public Domain Information About Windows Sockets

PSS ID Number: Q115045

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0
- Microsoft Windows SDK for Windows, version 3.1

SUMMARY

=======

Anyone with FTP access to the Internet can obtain a vast amount of information pertinent to the Windows Sockets API. The information is available through anonymous FTP to sunsite.unc.edu in the /pub/micro/pc-stuff/ms-windows/winsock directory. In the winsock directory, there is a wealth of technical information, including specifications, applications, DLLs for testing, and developers' guides not only from Microsoft, but also from other vendors of Windows Sockets.

The following list gives a brief overview of the information contained in the directory. This information can also be obtained from the READ.ME file in the location mentioned above.

FAQ

A list of frequently asked questions, with companion answers. Please send additional questions or answers to towfiq@Microdyne.COM.

apps/

Applications that have been written to run on top of the Windows Sockets DLL. See the file READ.ME in the winsock directory for a description of each application.

./incoming/

A directory where submissions to the archive (such as sample programs) should be placed.

packages/

The winsock directory contains different vendors' WINSOCK.DLLs for testing purposes as well as Windows Sockets development packages. See the file READ.ME in this directory for a description of each package.

press-releases/

Some of the press releases about the Windows Sockets.

winsock-archive/

An archive of the winsock@Sunsite.UNC.Edu mailing list. (Send subscription requests to listserv@SunSite.UNC.Edu.)

wsguide.doc

The "Windows Sockets Guide," by Martin Hall, JSB (martinh@jsbus.com) in Microsoft Word for Windows format.

wsguide.ps

The "Windows Sockets Guide," by Martin Hall, JSB (martinh@jsbus.com) in PostScript format. In the subdirectories winsock-1.0 and winsock-1.1, you can find the specification in many different formats.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbref
KBSubCategory: NtwkMisc

ODBC, MAPI, Video for Windows, NetDDE, & RAS/RPC Under Win32s

PSS ID Number: Q147432

The information in this article applies to:

- Microsoft Win32s version 1.3

The following interfaces are not completely supported under Win32s. Any exceptions are noted.

- RAS API and RPC.

- Video for Windows API (note that the MCI interface is supported).

- MAPI interface (Also, a Win32s client can not thunk down to 16-bit MAPI in Windows 3.x. You will have to write a 16-bit client that calls the 16-bit MAPI APIs, and then have the Win32s client thunk to this 16-bit client).

- NetDDE interface.

- ODBC (Win32s supports only ODBC version 2.0 and 2.1. Later versions of ODBC were not completely tested on Win32s).

Additional reference words: 1.30 kbinf
KBCategory: kbinterop
KBSubcategory: w32s

OFN_EXPLORER from 32-bit Thunk Brings Up Old-Style CommDlg

PSS ID Number: Q140726

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95

SUMMARY

You can have a 16-bit application running under Windows 95 take advantage of the new Explorer-style FileOpen common dialog by thunking up to a 32-bit DLL, specifying the OFN_EXPLORER style from that DLL, and linking it with the new COMDLG32.DLL. This, however, causes the old-style dialog to come up, instead of the new Explorer-style dialog.

MORE INFORMATION

Although the common dialog function is called from the 32-bit DLL, COMDLG32.DLL detects that the current process is still a 16-bit application and quickly reverts back to the old-style File Open common dialog. This is because the new Explorer-style dialog uses multiple threads, which 16-bit applications do not support.

The only way for a 16-bit application to take advantage of the new Explorer-style functionality is to port to Win32 and make the call to GetOpenFileName() directly.

Additional reference words: 4.00 kbinf
KBCategory: kbprg
KBSubcategory: UsrCmnDlg

Open File Dialog Box -- Pros and Cons

PSS ID Number: Q74612

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

Applications can call the common dialog library (COMMDLG.DLL) function GetOpenFileName() to retrieve the name of a file that the user wants to manipulate. Using this DLL provides a common interface for opening files across applications and also eliminates dialog-box message processing within the application's code. However, the application must initialize a structure specific to the dialog box. This article discusses the benefits and costs of using the Open File common dialog, via GetOpenFileName().

MORE INFORMATION

When an application uses the Open File dialog box provided by the common dialog DLL, the primary benefit to that application's users will be a familiar interface. Once they learn how to open a file in one application that uses the DLL, they will know how to open a file in all applications that use it.

Features of the Open File dialog include:

- A list box of files, filtered by extension, in the current directory
- A list box of directories from root of current drive to current directory, plus subdirectories
- A combo box of file types to list in filename list box
- A combo box of drives available, distinguished by drive type
- An optional "Read Only" check box
- An optional "Help" button
- An optional application hook function to modify standard behavior
- An optional dialog template to add private application features

For the application's programming staff, the benefits of using the Open File common dialog will include:

- No dialog-box message processing necessary to implement the Open File dialog box
- Drive and directory listings are constructed by the DLL, not by the application
- A full pathname for the file to be opened is passed back to the application, and this name can be passed directly to the OpenFile

function
- Offsets into the full pathname are also returned, giving the application access to the filename (sans pathname) and the file extension without the need for parsing
- The application can pass in its own dialog box template, in which case the DLL will use that template instead of the standard template
- The application can provide a dialog hook function to extend the interface of the DLL or to change how events are handled
- The application can choose to have a single filename or multiple filenames returned from the dialog box

The cost for the programming staff could be in adapting previously written application code to handle the common dialog interface. While making this change is straightforward, it does require coding time. If the application only needs a filename with no path, the Open File common dialog is probably not appropriate.

For more information on using the Open File common dialog, query on GetOpenFileName().

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCmnDlg

OpenComm() and Related Flags Obsolete Under Win32

PSS ID Number: Q94990

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

OpenComm(), a Windows 3.1 application programming interface (API), is obsolete under Windows NT and is not in the Win32 API. Note that the flags, IE_BADID, IE_BAUDRATE, IE_BYTESIZE, IE_DEFAULT, IE_HARDWARE, IE_MEMORY, IE_NOPEN, and IE_OPEN are obsolete, but are still in the header files.

OpenComm() is provided for 16-bit Windows-based applications running under Windows on Win32 (WOW).

MORE INFORMATION
================

Under Win32, CreateFile() is used to create a handle to a communications resource (for example, COM1). The fdwShareMode parameter must be 0 (exclusive access), the fdwCreate parameter must be OPEN_EXISTING, and the hTemplate parameter must be NULL. Read, write, or read/write access can be specified and the handle can be opened for overlapped I/O.

ReadFile() and WriteFile() are used for communciations I/O. The TTY sample program shipped with the Win32 Software Development Kit (SDK) demonstrates how to do serial I/O in a Win32-based application.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseCommapi

Overlapping Controls Are Not Supported by Windows

PSS ID Number: Q79981

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

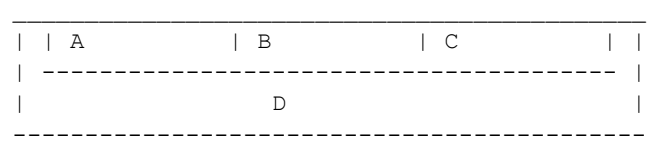
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

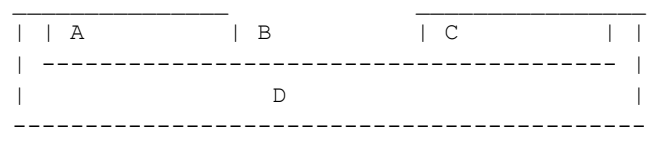
Child window controls should not be overlapped in applications for the Windows operating system. When one control overlaps another control, or another child window, the borders shared by the controls may not be drawn properly. Overlapping controls may confuse the user of the application because clicking the mouse in the common area may not activate the control that the user intended to activate. This behavior is a consequence of the way that Windows is designed.

MORE INFORMATION
================

The following example illustrates the painting problems caused by the ambiguity of overlapping borders. Consider three edit controls, called A, B and C, which overlap each other, and an enclosing child window D:



Assume that control B has the focus. If this set of controls is covered by another window, which is subsequently moved away, Windows will send a series of client and nonclient messages to each of the controls and to the enclosing child window. The result of these messages may appear as the illustration below, where the portion of window B's border that overlapped with part of window D's border is missing:



Repainting problems related to overlapping controls may vary depending on the version of Windows used.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 CS_PARENTDC
WS_CLIPCHILDREN
WS_CLIPSIBLINGS WM_NCPAINT WM_PAINT
KBCategory: kbui
KBSubcategory: UsrDlgs

Overview of the Windows 95 Virtual Address Space Layout

PSS ID Number: Q125691

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

=======

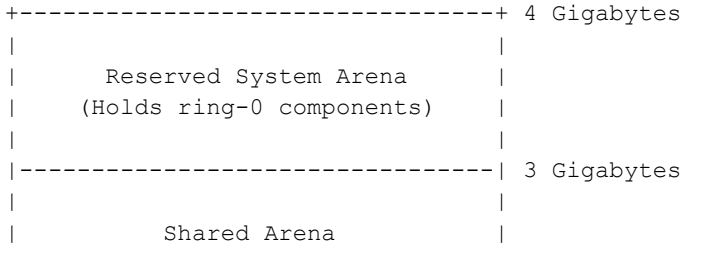
The virtual memory management mechanism in Microsoft Windows 95 makes it possible to execute Win32-based, 16-bit-based, and MS-DOS-based applications simultaneously. To accomplish this, the virtual memory manager uses a virtual address space layout that is considerably different from that used by Microsoft Windows version 3.x and that is slightly different from that used by Microsoft Windows NT. Although the differences from Windows NT are slight, they are important.

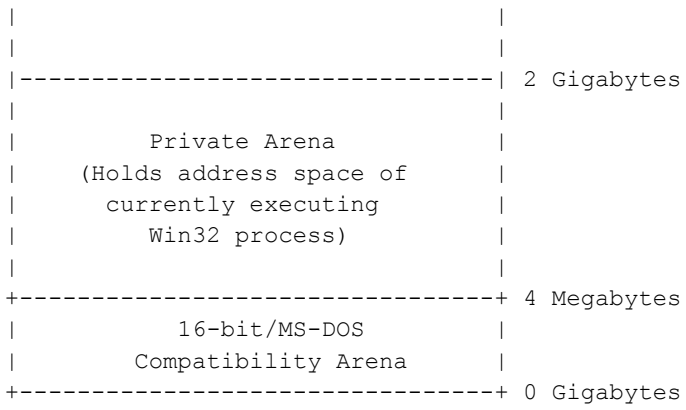
The memory manager in Windows 95 uses paging and 32-bit linear addressing to provide a full 32-bit virtual address space that has a maximum size of four gigabytes (GB). This four-GB address space is partitioned by the memory manager into four major sections, known as arenas, that are used for different types of applications and parts of the system. The first arena, from zero to four megabytes (MB) exists for compatibility with applications based on Windows version 3.1x and MS-DOS. The next arena, from four MB to two GB, is the private address space for each Win32 process. The third arena, from two to three GB, is a shared address space that contains memory mapped files and the 16-bit components. Finally, the fourth arena, from three to four GB, is reserved for the system's use.

MORE INFORMATION

================

The following diagram shows the overall virtual address space layout used in Windows 95. The Compatibility Arena holds the current virtual machine and other software. Each Win32 process gets its own private address space in which to execute. The Private Arena contains the currently executing Win32 process's private address space. All 16-bit-based applications and DLLs, including the 16-bit Windows system components, reside in the Shared Arena. Finally, the Reserved System Arena is used to store all ring-0 code such as the virtual machine manager and virtual device drivers. It is not accessible by either 16-bit-based or Win32-based applications.





Each arena has a specific purpose and is described in detail below.

16-bit/MS-DOS Compatibility Arena

The first four megabytes of the system's address space is reserved by the system and is accessible to 16-bit and MS-DOS software for compatibility. The current virtual machine occupies the lowest megabyte of this area. The remaining three megabytes are mostly empty space but may contain MS-DOS device drivers and Terminate & Stay Resident (TSR) programs.

The 16-bt/MS-DOS Compatibility Arena is not accessible to Win32 processes for reading or writing. This means Win32 processes may may not allocate memory, load DLLs, or be loaded below the four megabyte (MB) address.

Private Arena

The private arena holds the private address space of the currently executing Win32 process. Because every Win32 process gets its own address space, the contents of this arena will depend upon which process is currently executing. The memory manager maps the pages of a process's private address space so that other processes cannot access it and corrupt the process. The process's code, data, and dynamically-allocated memory all exist in the private address space.

With the exception of the system's shared DLLs (USER32.DLL, GDI32.DLL, and KERNEL32.DLL), all DLLs loaded by the process are mapped into the process's private address space. Windows extension DLLs such as SHELL32.DLL, COMCTL32.DLL, and COMDLG32.DLL are not system shared DLLs and are mapped into the process's private address space.

Because console applications are Win32-based applications without graphical user interfaces, they too get their own private address spaces, as do Win32 graphical user interface (GUI) applications.

The minimum load address for a Win32 process in Windows 95 is four MB because the first four megabytes are reserved for the Compatibility Arena.

Shared Arena

The shared arena is unique to Windows 95. This arena contains components that must be mapped into every process's address space. All of the pages in this arena are mapped identically in every process.

The 16-bit global heap, which contains all 16-bit-based applications, DLLs, and 16-bit system DLLs, resides in the shared arena. The Win32 shared system DLLs (USER32.DLL, GDI32.DLL, and KERNEL32.DLL) are also located in the shared arena.

Unlike the Reserved System Arena, the shared arena is readable and writable by Win32 and 16-bit processes alike. This doesn't mean they are free to get memory directly from this address space. All 16-bit-based applications and DLLs actually are located in the 16-bit global heap, so they allocate memory from the 16-bit global heap; when this heap needs to be grown, KRNL386.EXE gets the memory from the shared arena.

Win32 processes may not allocate memory directly from the shared arena, but they always use it for mapping views of file mappings. Unlike Windows NT, where views of file mappings always are placed in the private address space, Windows 95 holds views of file mappings in the shared arena.

The DOS Protected Mode Interface (DPMI) server's memory pool is located in the Shared Arena. Thus, calls to the DPMI server to allocate memory will result in memory that is globally accessible.

Sometimes, a virtual device driver (VxD) may need to map a buffer passed to it by a Win32 process into globally accessible memory so that the buffer can be accessed even if the process isn't in context. By calling _LinPageLock virtual machine manager service with the PAGEMAPGLOBAL flag, a VxD can obtain a linear address in the shared arena that corresponds to the buffer passed to it by the Win32 process.

Reserved System Arena

The reserved system contains the code and data of all ring-0 components such as the virtual machine manager, DOS extender, DPMI server, and virtual device drivers. This arena is used exclusively by ring-0 components and not addressable by ring 3 code, such as MS-DOS-based, 16-bit-based, and Win32-based applications and DLLs.

Additional reference words: 4.00 layout memory virtual
KBCategory: kbprg
KBSubcategory: BseMm

Owner-Draw Buttons with Bitmaps on Non-Standard Displays

PSS ID Number: Q67715

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

If an application contains an owner-draw button that paints itself with a bitmap, the application's resources must contain a set of bitmaps appropriate to each display type on which the application might run.

If the application's resources do not contain bitmaps suitable for the display on which the application is running, the application can use the default 3-D button appearance by changing the button style to BS_PUSHBUTTON from BS_OWNERDRAW.

Under Windows 95, there is a new style BS_BITMAP which applications might find easier to use.

Changing the style of a button is possible in Windows version 3.0; however, this technique is not guaranteed to be supported in future releases of Windows.

MORE INFORMATION

An owner-draw button can use bitmaps to paint itself. When an application contains this type of owner-draw button, it must also contain a set of bitmaps appropriate for each display type on which the application might run. Each set of bitmaps has a normal "up" bitmap and a depressed "down" bitmap to implement the 3-D effects. The most common standard Windows display types are: CGA, EGA, VGA, 8514/a, and Hercules Monochrome. The dimensions and aspect ratio of the display affect the appearance of the bitmap. For example, a monochrome bitmap designed for VGA will display correctly on an 8514/a and any other display with a 1:1 aspect ratio.

If an application determines that it does not contain an appropriate set of bitmaps for the current display type, then it should change the button style from BS_OWNERDRAW to BS_PUSHBUTTON. After the style has been changed and the button has been redrawn, the button will appear as a normal 3-D push button.

The following code fragment demonstrates how to change the style of a push button from owner-draw to normal:

```
    ...

    /*
     * hWndButton is assumed to be the handle to the button.
     * Note that lParam has a nonzero value, which forces the button
     * to be redrawn. This assures that the normal button appearance
     * will show after this message is sent.
     */
    SendMessage(hWndButton, BM_SETSTYLE, BS_PUSHBUTTON, 1L);

    ...
```

Additional reference words: 3.00 3.10 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Owner-Draw: Overview and Sources of Information

PSS ID Number: Q64327

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Owner-draw controls are a new feature of Windows version 3.0. Because your application does all the drawing of the contents of the controls, you can customize them any way you like. Owner-draw controls are similar to predefined controls in that Windows will handle the control's functionality and mouse and keyboard input processing. However, you are responsible for drawing the owner-draw control in its normal, selected, and focus states.

You can create owner-draw controls from the menu, button, and list-box classes. You can create owner-draw combo boxes, but they must have the CBS_DROPDOWNLIST style (equates to a static text item and a list box). The elements of an owner-draw control can be composed of strings, bitmaps, lines, rectangles, and other drawing functions in any combination, in your choice of colors.

MORE INFORMATION

The Windows SDK sample application MENU demonstrates owner-draw menu items. The SDK sample application OWNCOMBO is a fairly large example of owner-draw and predefined list boxes and combo boxes.

The Microsoft Software Library contains simplified examples of an owner-draw push button, owner-draw list boxes, and an owner-draw drop-down list style combo box. Each of these examples includes descriptive text in a related Knowledge Base article. For additional information, please see the following article(s) in the Microsoft Knowledge Base:

ARTICLE-ID: Q64326
TITLE : Owner-Draw: Handling WM_DRAWITEM for Drawing Controls

-or-

ARTICLE-ID: Q64328
TITLE : Owner-Draw: 3-D Push Button made from Bitmaps with Text

-or-

ARTICLE-ID: Q65792

TITLE : Owner-Draw Example: Right - and Decimal Alignment

To respond to a WM_MEASUREITEM message, you MUST specify the height of the appropriate item in your control. Optionally, you can specify the item's width as well.

If you want to do something special when a string is deleted from a list box, you should process WM_DELETEITEM messages. Windows's default action is to erase the deleted string and to redraw the list box.

If you want to have control over the sorting for the order of items in a list box or combo box that does not have the *_HASSTRINGS style, you should specify the appropriate *_SORT style and process WM_COMPAREITEM messages. If the *_SORT and *_HASSTRINGS styles are present, Windows will automatically do the sorting without sending WM_COMPAREITEM messages. If *_SORT is not specified, WM_COMPAREITEM messages will not be generated and items will be displayed in the list box in the order in which they were inserted.

The heart of owner-draw controls is the response to WM_DRAWITEM messages. During this processing is when you draw an entire button or each individual item in a menu, list box, or combo box. Because Windows does not interfere in the drawing of owner-draw controls, your application must draw the specified control item. The display of the control must indicate the state of the control. Common states are as follows:

1. Focus state (has the focus or not)

2. Selection state (selection or not)

3. Emphasis state (active, grayed, or disabled) (less common to process)

See the article titled "Owner-Draw: Handling WM_DRAWITEM for Drawing Controls" for information about drawing controls in their various states. Familiarity with the WM_DRAWITEM message and the various control states is extremely helpful before trying to follow the code examples.

Under Windows 95, the BS_BITMAP style allows buttons to display bitmaps without using owner-draw.

Additional reference words: 3.00 3.10 3.50 4.00 owndraw od owner draw
KBCategory: kbui
KBSubcategory: UsrCtl

Owners Have Special Access to Their Objects

PSS ID Number: Q130543

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5 and 3.51

The Windows NT operating system allows the owner of an object to determine what types of access are granted or denied for a given user. This is referred to as Discretionary Access Control (DAC). In addition to granting the generic read and write types of access, the owner of an object can also grant other users the right to modify the access allowed to the object.

The access right to view the access allowed on an object is called READ_CONTROL. This is often granted as part of a generic right. The access right that allows someone to change the access for an object is called WRITE_DAC.

The owner of an object can always request WRITE_DAC and READ_CONTROL access to the object. This prevents a situation where the owner of an object can not manipulate the object. This also allows owners of objects to restrict their own access to the object (to guard against accidents) without having to explicitly grant READ_CONTROL and WRITE_DAC access to their accounts.

Additional reference words: 3.10 3.50 AccessCheck
KBCategory: kbprg
KBSubcategory: BseSecurity

PAGE_READONLY May Be Used as Discardable Memory

PSS ID Number: Q94947

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

Virtual memory pages marked as PAGE_READONLY under Win32 may be used the way discardable segments of memory are used in Windows 3.1. These virtual memory pages are by default not "dirty," so the system may use them (zeroing them first if necessary) without having to first write their contents to disk.

From a system resource perspective, PAGE_READONLY is treated similar to discardable memory under Windows 3.1 when the system needs to free up resources. From a programming standpoint, the system automatically re-reads the memory when the data is next accessed (for example, we attempt to access our page when it has been "discarded," a page fault is generated, and the system reads it back in). Memory-mapped files are convenient for setting up this type of behavior.

If a PAGE_READONLY memory page becomes dirty [by changing the protection via VirtualProtect() to PAGE_READWRITE, changing the data, and restoring PAGE_READONLY], the page will be written to disk before the system uses it.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMm

Panning and Scrolling in Windows

PSS ID Number: Q11619

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

When using bitmaps, the mapping mode is ignored and physical units (in other words, MM_TEXT pixels) are used. It is not necessary to use the extent/origin routines to keep track of the logical origin.

If scrolling is desired and if there are no child windows in the client area, it is best to BitBlt the client area to scroll it, and PatBlt the uncovered area with the default brush.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrPnt

Passing a Pointer to a Member Function to the Win32 API

PSS ID Number: Q102352

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Many of the Win32 application programming interfaces (APIs) call for a callback routine. One example is the lpStartAddr argument of CreateThread():

```
HANDLE CreateThread(lpsa, cbStack, lpStartAddr, lpvThreadParm,
   fdwCreate, lpIDThread)

LPSECURITY_ATTRIBUTES lpsa;    /* Address of thread security attrs */
DWORD cbStack;                    /* Initial thread stack size*/
LPTHREAD_START_ROUTINE lpStartAddr; /* Address of thread function */
LPVOID lpvThreadParm;       /* Argument for new thread*/
DWORD fdwCreate;               /* Creation flags*/
LPDWORD lpIDThread;               /* Address of returned thread ID */
```

When attempting to use a member function as the thread function, the following error is generated:

```
error C2643: illegal cast from pointer to member
```

The problem is that the function expects a C-style callback, not a pointer to a member function. A major difference is that member functions are called with a hidden argument called the "this" pointer. In addition, the format of the pointer isn't simply the address of the first machine instruction, as a C pointer is. This is particularly true for virtual functions.

If you want to use a member function as a callback, you can use a static member function. Static member functions do not receive the "this" pointer and their addresses correspond to an instruction to execute.

Static member functions can only access static data, and therefore to access nonstatic class members, the function needs an object or a pointer to an object. One solution is to pass in the "this" pointer as an argument to the member function.

MORE INFORMATION

================

This situation occurs with callback functions of other types as well, such as:

```
   DLGPROC                  GRAYSTRINGPROC
   EDITWORDBREAKPROC        LINEDDAPROC
   ENHMFENUMPROC            MFENUMPROC
   ENUMRESLANGPROC          PROPENUMPROC
   ENUMRESNAMEPROC          PROPENUMPROCEX
   ENUMRESTYPEPROC          TIMERPROC
   FONTENUMPROC             WNDENUMPROC
   GOBJENUMPROC
```

For more information on C++ callbacks, please see the May 1993 issue of the "Windows Tech Journal."

The following sample demonstrates how to use a static member function as a thread function, and pass in the "this" pointer as an argument.

Sample Code

```
#include <windows.h>

class A
{
public:
   int x;
   int y;

   A() { x = 0; y = 0; }

   static StartRoutine( A * );   // Compiles clean, includes "this" pointer
};

void main( )
{
   A a;

   DWORD dwThreadID;

   CreateThread( NULL,
      0,
      (LPTHREAD_START_ROUTINE)(a.StartRoutine),
      &a,                        // Pass "this" pointer to static member fn
      0,
      &dwThreadID
   );
}
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

PEN2CTL.VBX Custom Controls for Windows 95 Pen Services

PSS ID Number: Q130654

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) Version 4.0

SUMMARY

=======

The PEN2CTL.VBX provides three pen edit custom controls to allow for quick development of pen-aware applications using Microsoft Windows 95 Pen Services. The application designer may substitute these custom controls for the standard input controls available in Visual Basic.

MORE INFORMATION

================

The PEN2CTL.VBX ships with version 4.0 of the Win32 SDK and is available in the Microsoft Software. It is installed along with the Pen 2.0 Samples Library.

To get PEN2CTL.VBX along with the PENVBX.HLP Help file and the sample application files discussed in this article, download PEN2CTL.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download PEN2CTL.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get PEN2CTL.EXE

The PEN2CTL.EXE self-extracting file contains:

- PENVBX.HLP, a help file that fully explains all the technical issues and describes how to interact with these pen-aware custom controls.

- Several files that comprise a sample Pen application that demonstrates how to use all the major features of the PEN2CTL.VBX.

- PEN2CTL.VBX, which is freely redistibutable and implements the three pen-aware custom edit controls: HEdit, BEdit, and lEdit.

  - HEdit is the handwriting edit control. It accepts free-form input. This control is similar to the standard Visual Basic text box except data can be entered using a pen in addition to the normal keyboard input method.

  - BEdit is the boxed edit control. It expands the properties of the

handwriting edit control. It allows for additional manipulation of the writing area and provides the application with comb or box guides that accept pen input. Each segment or box accepts only a single character of input. This increases the accuracy of the recognition and in most cases is preferable to the handwriting edit control. The BEdit control also has the ability to provide a list of alternate words from which you may choose.

- lEdit is the pen ink edit control. It is similar to a picture box control in that it allows you to draw, erase, move, resize, manipulate, and format pen strokes (called ink) on the control. It also allows you to set background pictures and grid lines as well as create bitmaps of the background and/or ink.

While there are many similarities between the controls provided by the Pen 1.0 VBX (PENCNTRL.VBX) and those provided by PEN2CTL.VBX, there are also many differences. Those familiar with the Pen 1.0 VBX should refamiliarize themselves with PEN2CTL.VBX and its changes.

NOTE: Neither VBX is compatible with the other but both implement some of the same custom controls, so developers should make sure the correct VBX is installed in their development environment and is distributed with their application.

Additional reference words: 4.00 2.00
KBCategory: kbprg kbfile
KBSubcategory: WpenMisc WpenVB

Performing a Clear Screen (CLS) in a Console Application

PSS ID Number: Q99261

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

There is no Win32 application programming interface (API) that will clear the screen in a console application. However, it is fairly easy to write a function that will programmatically clear the screen.

MORE INFORMATION

The following function clears the screen:

```
void cls( HANDLE hConsole )
{
   COORD coordScreen = { 0, 0 };    /* here's where we'll home the
                                       cursor */
   BOOL bSuccess;
   DWORD cCharsWritten;
   CONSOLE_SCREEN_BUFFER_INFO csbi; /* to get buffer info */
   DWORD dwConSize;                 /* number of character cells in
                                       the current buffer */

   /* get the number of character cells in the current buffer */

   bSuccess = GetConsoleScreenBufferInfo( hConsole, &csbi );
   PERR( bSuccess, "GetConsoleScreenBufferInfo" );
   dwConSize = csbi.dwSize.X * csbi.dwSize.Y;

   /* fill the entire screen with blanks */

   bSuccess = FillConsoleOutputCharacter( hConsole, (TCHAR) ' ',
      dwConSize, coordScreen, &cCharsWritten );
   PERR( bSuccess, "FillConsoleOutputCharacter" );

   /* get the current text attribute */

   bSuccess = GetConsoleScreenBufferInfo( hConsole, &csbi );
   PERR( bSuccess, "ConsoleScreenBufferInfo" );

   /* now set the buffer's attributes accordingly */

   bSuccess = FillConsoleOutputAttribute( hConsole, csbi.wAttributes,
      dwConSize, coordScreen, &cCharsWritten );
```

```
   PERR( bSuccess, "FillConsoleOutputAttribute" );

   /* put the cursor at (0, 0) */

   bSuccess = SetConsoleCursorPosition( hConsole, coordScreen );
   PERR( bSuccess, "SetConsoleCursorPosition" );
   return;
}
```

Additional reference words: 3.10 3.50 4.00 95 clearscreen
KBCategory: kbprg
KBSubcategory: BseCon

Performing a Synchronous Spawn Under Win32s

PSS ID Number: Q125212

The information in this article applies to:

- Microsoft Win32s, version 1.2

SUMMARY

=======

Under Windows NT, you can synchronously spawn an application (that is, spawn an application and wait until the spawned application is terminated before continuing). To do so, call CreateProcess() to start the application, and pass the handle returned to WaitForSingleObject() to wait for the application to terminate. This is shown in the sample code in the "More Information" section in this article.

However, this method does not work under Win32s. Under Win32s, CreateProcess() does not return the process handle for 16-bit Windows-based applications, only for Win32-based application. Even if it did, the method described in the proceeding paragraph would not work under Win32s because WaitForSingleObject() returns TRUE immediately under Win32s.

In fact, there is no 32-bit only solution for this issue. The 32-bit WinExec() does not return an instance handle as the 16-bit WinExec() does. In addition, you cannot use GetExitCodeProcess() to find the exit status of 16-bit Windows-based applications in order to loop on their status. It is a limitation that GetExitCodeProcess() returns zero for 16-bit Windows-based applications on both Windows NT and Win32s.

The solution is to create a thunk to the 16-bit side and from the 16-bit side, solve the problem as you would normally solve it from a Windows-based application. Namely, start the application with WinExec() and use one of the Toolhelp APIs in a test loop to determine when the application is terminated. Alternatively, you can use EnumWindows() to determine when the application is terminated. The sample code below uses the Toolhelp APIs.

MORE INFORMATION

================

Sample code to perform a synchronous spawn is given below. The code is divided into three source files:

- The main application.
- The 32-bit side of the thunk.
- The 16-bit side of the thunk.

You can use the thunking code as is, calling SynchSpawn() in your own application as demonstrated in the main application below. For information on Universal thunks (including which header files and libraries to use), please see the "Win32s Programmer's Reference."

In all three modules, use the following header file SPAWN.H:

```
/*** Function Prototypes ****/

DWORD APIENTRY SynchSpawn( LPCSTR lpszCmdLine, UINT nCmdShow );

/*** Constants for Dispatcher ***/

#define SYNCHSPAWN     1
```

Main Application

This application attempts to synchronously spawn NOTEPAD under Windows NT and Win32s. NOTE: Under Win32s, NOTEPAD is a 16-bit application.

GetVersion() is used to detect the platform. Under Windows NT, CreateProcess() and WaitForSingleObject() perform the spawn. Under Win32s, the thunked routine SynchSpawn() is called.

```
/*** Main application code ***/

#include <windows.h>
#include "spawn.h"

void main()
{
   DWORD dwVersion;
   STARTUPINFO si = {0};
   PROCESS_INFORMATION pi = {0};

   dwVersion = GetVersion();

   if( !(dwVersion & 0x80000000) )  // Windows NT
   {
      si.cb = sizeof(STARTUPINFO);
      si.lpReserved = NULL;
      si.lpReserved2 = NULL;
      si.cbReserved2 = 0;
      si.lpDesktop = NULL;
      si.dwFlags = 0;

      CreateProcess( NULL,
                     "notepad",
                     NULL,
                     NULL,
                     TRUE,
                     NORMAL_PRIORITY_CLASS,
                     NULL,
                     NULL,
                     &si,
                     &pi );
      WaitForSingleObject( pi.hProcess, INFINITE );
   }
   else if( LOBYTE(LOWORD(dwVersion)) < 4 )  // Win32s
   {
```

```
      SynchSpawn( "notepad.exe", SW_SHOWNORMAL );
   }

   MessageBox( NULL, "Return from SynchSpawn", " ", MB_OK );

}
```

32-bit Side of Thunk

This DLL provides the 32-bit side of the thunk. If the DLL is loaded under Win32s, it initializes the thunk in its DllMain() by calling UTRegister(). The entry point SynchSpawn() packages up the arguments and calls the 16-bit side through the thunk.

```
/*** Code for 32-bit side of thunk ***/

#define W32SUT_32    // Needed for w32sut.h in 32-bit code

#include <windows.h>
#include "w32sut.h"
#include "spawn.h"

typedef BOOL (WINAPI * PUTREGISTER) ( HANDLE     hModule,
                  LPCSTR     lpsz16BitDLL,
                  LPCSTR     lpszInitName,
                  LPCSTR     lpszProcName,
                  UT32PROC * ppfn32Thunk,
                  FARPROC    pfnUT32Callback,
                  LPVOID     lpBuff
                );

typedef VOID (WINAPI * PUTUNREGISTER) (HANDLE hModule);

typedef DWORD (APIENTRY *PUT32CBPROC) (LPVOID lpBuff, DWORD dwUserDefined);

UT32PROC      pfnUTProc = NULL;
PUTREGISTER   pUTRegister = NULL;
PUTUNREGISTER pUTUnRegister = NULL;
PUT32CBPROC   pfnUT32CBProc = NULL;
int           cProcessesAttached = 0;
BOOL          fWin32s = FALSE;
HANDLE        hKernel32 = 0;

/*****************************************************************\
* Function: BOOL APIENTRY DllMain(HANDLE, DWORD, LPVOID)          *
*                                                                 *
*  Purpose: DLL entry point. Establishes thunk.                   *

BOOL APIENTRY DllMain(HANDLE hInst, DWORD fdwReason, LPVOID lpReserved)
{
   DWORD dwVersion;

   if ( fdwReason == DLL_PROCESS_ATTACH )
   {
```

```
    /*
     * Registration of UT need to be done only once for first
     * attaching process.  At that time set the fWin32s flag
     * to indicate if the DLL is executing under Win32s or not.
     */

      if( cProcessesAttached++ )
      {
         return(TRUE);         // Not the first initialization.
      }

      // Find out if we're running on Win32s
      dwVersion = GetVersion();
      fWin32s = (BOOL) (!(dwVersion < 0x80000000))
                       && (LOBYTE(LOWORD(dwVersion)) < 4);

      if( !fWin32s )
         return(TRUE);         // Win32s - no further initialization needed

      hKernel32 = LoadLibrary( "Kernel32.Dll" ); // Get Kernel32.Dll handle

      pUTRegister = (PUTREGISTER) GetProcAddress( hKernel32, "UTRegister"
);


      if( !pUTRegister )
         return(FALSE);        // Error- Win32s, but can't find UTRegister

      pUTUnRegister = (PUTUNREGISTER) GetProcAddress(hKernel32,
                                                     "UTUnRegister");

      if( !pUTUnRegister )
         return(FALSE);        // Error- Win32s, but can't find
UTUnRegister


      return (*pUTRegister)( hInst,           // Spawn32.DLL module handle
                             "SPAWN16.DLL",   // 16-bit thunk dll
                             "UTInit",        // init routine
             "UTProc",         // 16-bit dispatch routine
                             &pfnUTProc,      // Receives thunk address
                             pfnUT32CBProc,   // callback function
                             NULL );          // no shared memroy

   }
   if((fdwReason==DLL_PROCESS_DETACH)&&(0==--cProcessesAttached)&&fWin32s)
   {
      (*pUTUnRegister)( hInst );
      FreeLibrary( hKernel32 );
   }
} // DllMain()

/*****************************************************************\
* Function: DWORD APIENTRY SynchSpawn(LPTSTR, UINT)                *
```

```
*                                                                     *
* Purpose: Thunk to 16-bit code                                       *

DWORD APIENTRY SynchSpawn( LPCSTR lpszCmdLine, UINT nCmdShow )
{
   DWORD Args[2];
   PVOID Translist[2];

   Args[0] = (DWORD) lpszCmdLine;
   Args[1] = (DWORD) nCmdShow;

   Translist[0] = &Args[0];
   Translist[1] = NULL;

   return( (* pfnUTProc)( Args, SYNCHSPAWN, Translist) );
}
```

16-bit Side of Thunk

This DLL provides the 16-bit side of the thunk. The LibMain() and WEP() of this 16-bit DLL perform no special initialization. The UTInit() function is called during thunk initialization; it stores the callback procedure address in a global variable. The UTProc() function is called with a code that indicates which thunk was called as its second parameter. In this example, the only thunk provided is for SynchSpawn(). The synchronous spawn is performed in the SYNCHSPAWN case of the switch statement in the UTProc().

NOTE: UTInit() and UTProc() must be exported. This can be done in the module definition (.DEF) file.

```
/* Code for 16-bit side of thunk.                              */
/* Requires linking with TOOLHELP.LIB, for ModuleFindHandle(). */

#ifndef APIENTRY
#define APIENTRY
#endif
#define W32SUT_16     // Needed for w32sut.h in 16-bit code

#include <windows.h>
#include <toolhelp.h>
#include <malloc.h>
#include "w32sut.h"
#include "spawn.h"

UT16CBPROC glpfnUT16CallBack;

/*********************************************************************\
* Function: LRESULT CALLBACK LibMain(HANDLE, WORD, WORD, LPSTR)       *
*                                                                     *
*  Purpose: DLL entry point                                           *

int FAR PASCAL LibMain( HANDLE hLibInst, WORD wDataSeg,
         WORD cbHeapSize, LPSTR lpszCmdLine)
```

```
{
   return (1);
} // LibMain()

/*********************************************************************\
* Function: DWORD FAR PASCAL UTInit(UT16CBPROC, LPVOID)               *
*                                                                     *
*  Purpose: Universal Thunk initialization procedure                  *

DWORD FAR PASCAL UTInit( UT16CBPROC lpfnUT16CallBack, LPVOID lpBuf )
{
   glpfnUT16CallBack = lpfnUT16CallBack;
   return(1);   // Return Success
} // UTInit()


/*********************************************************************\
* Function: DWORD FAR PASCAL UTProc(LPVOID, DWORD)                    *
*                                                                     *
*  Purpose: Dispatch routine called by 32-bit UT DLL                  *

DWORD FAR PASCAL UTProc( LPVOID lpBuf, DWORD dwFunc)
{
   switch (dwFunc)
   {
      case SYNCHSPAWN:
      {
          HMODULE hMod;
          MODULEENTRY FAR *me;
          UINT hInst;
          LPCSTR lpszCmdLine;
          UINT nCmdShow;
          MSG msg;
          BOOL again=TRUE;

          /* Retrieve the command line arguments stored in buffer */

          lpszCmdLine = (LPSTR) ((LPDWORD)lpBuf)[0];
          nCmdShow = (UINT) ((LPDWORD)lpBuf)[1];

          /* Start the application with WinExec() */

          hInst = WinExec( lpszCmdLine, nCmdShow );
          if( hInst < 32 )
             return 0;

          /* Loop until the application is terminated. The Toolhelp API
           * ModuleFindHandle() returns NULL when the application is
           * terminated. NOTE: PeekMessage() is used to yield the
           * processor; otherwise, nothing else could execute on the
           * system.
           * /

          hMod = GetModuleHandle( lpszCmdLine );
```

```
            me = (MODULEENTRY FAR *) _fcalloc( 1, sizeof(MODULEENTRY) );
            me->dwSize = sizeof( MODULEENTRY );
            while( NULL != ModuleFindHandle( me, hMod ) && again )
            {
               while( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) && again )
               {
                  if(msg.message == WM_QUIT)
                  {
                     PostQuitMessage(msg.wParam);
                     again=FALSE;
                  }
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
               }
            }
            return 1;
         }

      } // switch (dwFunc)

      return( (DWORD)-1L ); // We should never get here.
   } // UTProc()

   /*****************************************************************\
   * Function: int FAR PASCAL _WEP(int)                              *
   *                                                                 *
   *  Purpose: Windows exit procedure                                *

   int FAR PASCAL _WEP( int bSystemExit )
   {
      return (1);
   } // WEP()
```

Additional reference words: 1.20 win16
KBCategory: kbprg kbcode
KBSubcategory: W32s

PHONECAPS for Phones That Don't Report Button States

PSS ID Number: Q108308

The information in this article applies to:

- Microsoft Windows Telephony Software Development Kit (SDK) version 1.0 for Windows version 3.1
- Microsoft Win32 SDK, version 4.0

A Windows Telephony application can call phoneGetDevCaps to inquire about a phone's telephony capabilities. The resulting function in the TAPI (Telephony application programming interface) service provider is TSPI_phoneGetDevCaps. If the phone device does not report button states, it is acceptable for the service provider to set the dwButtonModeSize/Offset fields in the PHONECAPS structure to 0 (zero). Alternatively, the provider could describe the button(s) as PHONEBUTTONMODE_DUMMY.

As a clarification, setting the dwButtonModeSize to 0 or describing the buttons as PHONEBUTTON_DUMMY does not prohibit the service provider from setting and using the dwButtonFunctionsSize and dwButtonFunctionsOffset members of the PHONECAPS structure.

Additional reference words: 1.00 3.10 4.00 95
KBCategory: kbprg
KBSubcategory: MsgTapi

Physical Memory Limits Number of Processes/Threads

PSS ID Number: Q94840

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

Each time Windows NT creates an object, such as a process or a thread, it must allocate a certain amount of physical memory (nonpaged pool) for its support. The amount of memory that is needed for support of a process object is significantly higher than the memory requirement for support of a thread object. The amount of memory that is required for a thread object on a RISC machine is higher than the memory requirement for a thread on an x86 machine, due to the greater number and size of registers on the RISC machines.

Due to the physical memory requirement of processes and threads, programs that use the CreateProcess() and CreateThread() APIs should be careful to check their return codes to detect out-of-memory conditions.

On Windows NT 3.5, each time a process is created it reserves the minimum working set of memory. On a 32 MB system, the default minimum working set is 200 KB. Therefore, on a 32 MB system, you can create ~100 processes. You can lower your minimum working set to 80 KB (the lowest allowed) with the following call:

```
SetProcessWorkingSetSize( (HANDLE)(-1), 20*4096, 100*4096 );
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseProcThrd

Placing a Caret After Edit-Control Text

PSS ID Number: Q12190

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The EM_SETSEL message can be used to place a selected range of text in a Windows edit control. If the starting and ending positions of the range are set to the same position, no selection is made and a caret can be placed at that position. To place a caret at the end of the text in a Windows edit control and set the focus to the edit control, do the following:

```
   hEdit = GetDlgItem( hDlg, ID_EDIT );    // Get handle to control
   SetFocus( hEdit );
   SendMessage( hEdit, EM_SETSEL, 0, MAKELONG(0xffff,0xffff) );
```

It is also possible to force the caret to a desired position within the edit control. The following code fragment shows how to place the caret just to the right of the Nth character:

```
   hEdit = GetDlgItem( hDlg, ID_EDIT );    // Get handle to control
   SetFocus( hEdit );
   SendMessage( hEdit, EM_SETSEL, 0, MAKELONG(N,N) );
                                        // N is the character position
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Placing Captions on Control Windows

PSS ID Number: Q77750

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The SetWindowText() function can be used to place text into the caption bar specified for a control window. The control must have the WS_CAPTION style for the caption to be visible.

This technique does not work with edit controls because the SetWindowText() function specifies the contents of the edit control, not its caption.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Placing Double Quotation Mark Symbol in a Resource String

PSS ID Number: Q47674

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

To specify a set of double quotation marks within a string in an application's resource (RC) file, use two double quotation mark characters in succession, as in the following example:

Specify the following string in the RC file:

```
   "The letter ""Q"" is quoted."
```

The following string will appear in the compiled resource (RES) file:

```
   The letter "Q" is quoted.
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrRsc

Placing Text in an Edit Control

PSS ID Number: Q32785

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Text is placed into an edit control by calling SetDlgItemText() or by sending the WM_SETTEXT message to the edit control window, with lParam being a pointer to a null-terminated string. This message can be sent in two ways:

1. SendMessage(hwndEditControl, WM_SETTEXT, ...

2. SendDlgItemMessage(hwndParent, ID_EDITCTL, WM_SETTEXT...

   NOTE: hwndParent is the window handle of the parent, which may be a dialog or window. ID_EDITCTL is the ID of the edit control.

Text is retrieved from an edit control by calling GetDlgItemText() or by sending the WM_GETTEXT message to the edit control window, with wParam being the maximum number of bytes to copy and lParam being a far pointer to a buffer to receive the text. This message can be sent in two ways:

1. SendMessage(hwndEditControl, WM_GETTEXT, ...

2. SendDlgItemMessage(hwndParent, ID_EDITCTL, WM_GETTEXT...

   NOTE: hwndParent is the window handle of the parent, which may be a dialog or window. ID_EDITCTL is the ID of the edit control.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Points to Remember When Writing a Debugger for Win32s

PSS ID Number: Q121093

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, and 1.2

SUMMARY
=======

This article is intended for developers of debugging tools for the Win32s environment. It covers the issues that should be to taken into consideration while writing debugging tools for the Win32s environment.

Overall, the code of a debugger for the Win32s environment is similar to the code of a debugger for the Windows NT environment. There are special points that must be considered before and while writing debugging tools for the Win32s environment. They are:

- Using the WaitForDebugEvent() API Function
- Debugging Shared Code
- Getting and Setting Thread Context
- Tracing Through Mixed 16- and 32-bit Code
- Using Asynchronous Stops
- Identifying System DLLs
- Understanding Linear and Virtual Addresses
- Reading and Writing Process Memory
- Accessing the Thread Local Storage (TLS)

Each of these is discussed in detail in the More Information section below.

MORE INFORMATION
================

The following information is specific to writing a debugger for the Win32s environment.

Using the WaitForDebugEvent() API Function

The WaitForDebugEvent() API function waits for a debugging event to occur in a process being debugged. Use it to trap debugging events.

Because of the non-preemptive nature of Windows version 3.1, it is not possible to guarantee the timeout functionality. For this reason, the

dwTimeout parameter was implemented differently in Win32s. In Win32s, if dwTimeout is zero, the WaitForDebugEvent() function behaves as documented in the "Win32 Programmer's Reference." Otherwise, the function waits indefinitely, until a debug event occurs or until a message is received for that process.

Make sure the function returns if a message is received, so that the calling process can respond to messages. If WaitForDebugEvent() returns because a debug event has occurred, the return value is TRUE. Otherwise, the return value is FALSE. In Win32, a FALSE return value means failure. Have the calling process use SetLastError() to set the error value to 0 before calling WaitForDebugEvent(). Then if the return value is FALSE and error value returned by GetLastError() is still zero, it means a message arrived.

The following code fragment demonstrates the use of WaitForDebugEvent() in the message loop:

```
   while( GetMessage(&msg, NULL, NULL, NULL) )
   {
      TranslateMessage(&msg); /* Translate virtual key codes */
      DispatchMessage(&msg);  /* Dispatch message to window */

      SetLastError( 0 );      /* Set error code to zero  */
      if( WaitForDebugEvent(&DebugEvent, INFINITE) )
      {
      /* Process the debug event */
         ProcessDebugEvents( &DebugEvent );
      }
      else
      {
         if( GetLastError() != 0 )
         {
         /* Handle error condition. */
         }
      }
   }
```

Debugging Shared Code

Under Win32s, all processes run in a single address space. For that reason, if a debugger sets a breakpoint in shared code, all processes will encounter this breakpoint, even those that are not being debugged. For these processes, the debugger should restore the code, let the process execute the restored instruction, and then reset the breakpoint. The problem is that in order to do these operations, the debugger needs a handle to the process thread.

The debugger does not have a handle for the process thread of a process it did not create. To get the handle, Win32s supports a new function, OpenThread(), which is not a part of the Win32 API.

```
   HANDLE OpenThread(dwThreadId);
```

```
   DWORD dwThreadId;  /* The thread ID */

   Parameter description:

      dwThreadId - Specifies the thread identifier of the thread to open.

   Returns:

      If the function succeeds, the return value is an open handle of the
      specified thread; otherwise, it is NULL. To get extended error
      information, use the GetLastError() API.

   Comments:

      The handle returned by OpenThread() can be used in any function that
      requires a handle to a thread.
```

OpenThread() is exported by KERNEL32.DLL, but is not included in any of the SDK import libraries.

To create an import library on a Windows NT development machine:

1. Place the following contents into a file named W32SOPTH.C:

```
   #include <windows.h>

   HANDLE WINAPI OpenThread(DWORD dwThreadId)
   {
      return (HANDLE)NULL;
   }
```

2. Place the following contents into a file named W32SOPTH.DEF:

```
   LIBRARY kernel32

   DESCRIPTION 'Win32s OpenThread library'

   EXPORTS
      OpenThread
```

3. Place the following contents into a file named MAKEFILE:

```
   w32sopth.lib: w32sopth.obj
      lib -out:w32sopth.lib -machine:i386 -def:w32sopth.def w32sopth.obj

   w32sopth.obj: w32sopth.c
      cl /c w32sopth.c
```

4. Run the NMAKE utility from the directory that contains the files created in steps 1-3. This creates the W32SOPTH.LIB file.

The debugger should perform the following test: in the DEBUG_INFO structure returned by WaitForDebugEvent(), there is a thread ID. The debugger should check to see if this ID is one of the debugged processes. If it is not, the debugger should call OpenThread() with the given thread ID as the parameter

and receive a handle to the thread. Using this handle, the debugger should call GetThreadContext(), identify the breakpoint, restore the code, set the single step bit of EFlag, and resume the process by calling ContinueDebugEvent(). Then control returns to the debugger. The debugger restores the breakpoint. After dealing with the non-debugged process, the debugger must close the thread handle obtained from OpenThread() by using CloseHandle().

The following code fragment demonstrates how a debugger can handle breakpoints in the context of a non-debugged process:

```
   LPDEBUG_EVENT lpEvent;  /* Pointer to the debug event structure */
   HANDLE hProc;           /* Handle to process */
   HANDLE hThread;         /* Handle to thread */
   CONTEXT Context;        /* Context structure */;
   BYTE bOrgByte;          /* Original byte in the place of BP */
   DWORD cWritten;         /* Number of bytes written to memory */
   static DWORD dwBPLoc;   /* Breakpoint location */

   /*
    * Other debugger functions:
    *
    * LookupThreadHandle -
    *    Receives a thread ID and returns a handle to the thread, if
    *    the thread created by the debugger, else returns NULL.
    */
   HANDLE LookupThreadHandle(DWORD);

   /*
    * LookupOriginalBPByte -
    *    Receives an address of a breakpoint and returns the original
    *    contents of the memory in the place of the breakpoint.
    *    The memory contents is returned in the byte buffer passed as
    *    a parameter.
    * Return value - If the breakpoint was set by the debugger the
    *   return value is TRUE, else FALSE.
    */
   BOOL LookupOriginalBPByte( LPVOID, LPBYTE );

   /* Handle debug events according to event types */
   switch( lpEvent->dwDebugEventCode )
   {
   /* ... */
      case EXCEPTION_DEBUG_EVENT:
      /* Handle exception debug events according to exception type */
         switch( lpEvent->u.Exception.ExceptionRecord.ExceptionCode )
         {
         /* ... */
         case EXCEPTION_BREAKPOINT:
            /* Breakpoint exception */
            /* Look for the thread handle in the debugger tables */
               hThread = LookupThreadHandle( lpEvent->dwThreadId );
               if( hThread == NULL )
               {
               /* Not a debuggee */
```

```
      /* Get process and thread handles */
         hProc = OpenProcess( 0, FALSE, lpEvent->dwProcessId );
         hThread = OpenThread( lpEvent->dwThreadId );

      /* Get the full context of the processor */
         Context.ContextFlags = CONTEXT_FULL;
         GetThreadContext( hThread, &Context );

      /* We get the exception after executing the INT 3 */
         dwBPLoc = --Context.Eip;

      /* Restore the original byte in memory in the */

      /* place of the breakpoint */
         if( !LookupOriginalBPByte((LPVOID)dwBPLoc, &bOrgByte) )
         {
         /* Handle unfamiliar breakpoint */
         }
         else
         {
         /* Restore memory contents */
            WriteProcessMemory( hProc, (LPVOID)dwBPLoc,
            &bOrgByte, 1, &cWritten );

         /* Set the Single Step bit in EFlags */
            Context.EFlags |= 0x0100;
            SetThreadContext( hThread, &Context );
         }

      /* Free Handles */
         CloseHandle( hProc );
         CloseHandle( hThread );

      /* Resume the interrupted process */
         ContinueDebugEvent( lpEvent->dwProcessId,
                             lpEvent->dwThreadId, DBG_CONTINUE );
      }
      else
      {
      /* Handle debuggee breakpoint. */
      }
      break;

   case STATUS_SINGLE_STEP:
      hThread = LookupThreadHandle( lpEvent->dwThreadId );
      if( hThread == NULL )
      {
      /* Not a debuggee, just executed the original instruction */
      /* and returned to the debugger. */

      /* Get process handle */
         hProc = OpenProcess( 0, FALSE, lpEvent->dwThreadId );

      /* Restore the INT 3 instruction in the place of the BP */
         bOrgByte = 0xCC;
```

```
                WriteProcessMemory( hProc, (LPVOID)dwBPLoc,
                &bOrgByte, 1, &cWritten );

             /* Free Handle */
                CloseHandle( hProc );

             /* Resume the process */
                ContinueDebugEvent( lpEvent->dwProcessId,
                                    lpEvent->dwThreadId, DBG_CONTINUE );
             }
             else
             {
             /* Handle debuggee single-step. */
             }
             break;
       /* .... */
    }
 /* .... */
 }
```

This sample code does not contain code to handle error checking and return values from APIs. The assumption is that a non-debugged process generates a single step exception only when it is executing the instruction in the place of the breakpoint. The code for handling the single step exception does not handle debug registers.

Getting and Setting Thread Context

Because of architectural differences between Windows NT and Win32s, there is a difference in the way GetThreadContext() and SetThreadContext() work in Win32s. These functions return successfully only if they are called after returning from WaitForDebugEvent() with the EXCEPTION_DEBUG_EVENT value in the dwDebugEventCode field of the DEBUG_INFO structure and before calling ContinueDebugEvent(). At any other point, these APIs fail and GetLastError() returns ERROR_CAN_NOT_COMPLETE.

Tracing Through Mixed 16- and 32-bit Code

Occasionally, Win32-based applications switch to 16-bit mode and then go back to 32-bit mode. For example, part of the Windows API is implemented in Win32s by using thunks to connect to Windows version 3.1. That means that in order to call the API, Win32s switches to 16-bit mode, calls the corresponding API on the Windows version 3.1 side, and then returns to 32-bit mode.

Most debuggers do not allow tracing through 16-bit code. So when the code is about to switch to 16-bit mode, the debugger should trace over this code. To do so, Win32s supplies the DbgBackTo32 label. All calls to 16-bit code return through this address. The DbgBackTo32 label is exported by W32SKRNL.DLL. At this label, there is a RET instruction. After executing this RET instruction and immediately another following RET instruction, Win32s resumes execution at the application code, at the instruction following the call to the thunked function. So if the debugger determines

that the next call is into a thunk function, it can set a breakpoint at DbgBackTo32 and trace over this call.

Using Asynchronous Stops

The asynchronous stop key combination was set to CTRL+ALT+F11 in Win32s. It allows a 16-bit debugger to run at the same time as a 32-bit debugger. Each debugger can synchronously stop the other.

If the user presses CTRL+ALT+F11 when the executing code is 16-bit code, execution will not be interrupted until it returns to 32-bit code. This way, the debugger does not have to handle 16-bit code. If the user presses CTRL+ALT+F11 when the executing code is 32-bit code, execution is interrupted immediately.

Execution is interrupted by generating a single step exception. To handle the case where the user presses CTRL+ALT+F11 while 16-bit code is executing, the address of the exception is at a special Win32s label (W32S_BackTo32). This label is exported by W32SKRNL.DLL and is located a few instructions before DbgBackTo32. For more information on this see the "Tracing Through Mixed 16- and 32-bit Code" above.

The code at W32S_BackTo32 is system code and usually debuggers should not allow tracing through system code. But between W32S_BackTo32 and DbgBackTo32, the debugger may allow tracing through this specific code and also through the two following RET instructions. This will bring the user to the point in the application at which CTRL+ALT+F11 was pressed.

Identifying System DLLs

When tracing through application code, it is not desirable to trace into system DLL code. The main reason for this is that in many cases the code goes to 16-bit code. To enable the debugger to distinguish between system and user DLLs, all Win32s system DLLs contain an extra exported symbol called WIN32SYSDLL. The address of this symbol is meaningless. The existence of such a symbol indicates that this is a system DLL.

Understanding Linear and Virtual Addresses

Win32s uses flat memory address space as does Windows NT, but unlike Windows NT, the base of the code and data segments is not at zero. You must consider this when dealing with linear addresses -- such as hardware debug registers when setting a hardware breakpoint. When setting a hardware breakpoint, you need to add the base of the selector to the virtual address of the breakpoint and set the debug register with this value. If you do not do so, the code will run on Windows NT but not on Win32s.

The debugger needs to get the base address of the selectors by using the GetThreadSelectorEntry() function.

Similarly, when the hardware breakpoint is encountered, you must subtract the selector base address from the contents of the debug register in order

to read the process memory at the breakpoint location.

Reading and Writing Process Memory

When reading from or writing to process memory, all hardware breakpoints must be disabled. If you do not do so, accessing the memory locations pointed to by the debug registers will trigger the hardware breakpoints.

The following code demonstrates how a debugger can read process memory at the location of a read memory hardware breakpoint:

```
CONTEXT Context;
LDT_ENTRY SelEntry;
DWORD dwDsBase;
DWORD DR7;
BYTE Buffer[4];

/* Get Context */
Context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;
GetThreadContext( hThread, &Context );

/* Calculate the base address of DS */
GetThreadSelectorEntry(hThread, Context.SegDs, &SelEntry);
dwDsBase = ( SelEntry.HighWord.Bits.BaseHi << 24) |
             (SelEntry.HighWord.Bits.BaseMid << 16) |
             SelEntry.BaseLow;

/*
 * Disable all hardware breakpoints before reading the process
 * memory. Not doing so will lead to nested breapoints.
 */
DR7 = Context.Dr7;
Context.Dr7 &= ~0x3FF;
SetThreadContext( hThread, &Context );

/* Read DWORD at the location of DR0 */
ReadProcessMemory( hProcess,
                   (LPVOID)((DWORD)Context.Dr0-dwDsBase),
                   Buffer, sizeof(Buffer), NULL);

/* Restore hardware breakpoints */
Context.Dr7 = DR7;
SetThreadContext( hThread, &Context );
```

Accessing the Thread Local Storage (TLS)

The lpThreadLocalBase field of the CREATE_PROCESS_DEBUG_INFO structure in Windows NT specifies the base address of a per-thread data block. At offset 0x2C within this block, there exists a pointer to an array of LPVOIDs. There is one LPVOID for each DLL/EXE loaded at process initialization, and that LPVOID points to Thread Local Storage (TLS). This gives a debugger access to per-thread data in its debuggee's threads using the same

algorithms that a compiler would use.

On the other hand, in Win32s, lpThreadLocalBase contains a pointer directly to the array of LPVOIDs, not the pointer to the per-thread data block.

Additional reference words: 1.10 1.15 1.20
KBCategory: kbprg kbcode
KBSubCategory: W32s

Possible Cause for ERROR_INVALID_FUNCTION

PSS ID Number: Q111838

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

The Win32 Tape API (application programming interface) functions may return an error code of ERROR_INVALID_FUNCTION if the operation being attempted is not supported by the tape device.

MORE INFORMATION

================

You can determine what operations are valid for a tape device by calling GetTapeParameters() to get the tape drive parameters. See the code fragment in the Sample Code section of this article for an example of how to call GetTapeParameters() for this information.

Once you have the TAPE_GET_DRIVE_PARAMETERS structure, you can check for a specific operation in the FeaturesLow and FeaturesHigh members of the TAPE_GET_DRIVE_PARAMETERS structure. This is also demonstrated in the Sample Code section.

Sample Code

```
/*
** This is a code fragment only and will not compile and run as is.
*/

DWORD dwRes, dwSize;
TAPE_GET_DRIVE_PARAMETERS parmDrive;

...

dwSize = sizeof(parmDrive);
dwRes = GetTapeParameters(hTape, GET_TAPE_DRIVE_INFORMATION,
                &dwSize, &parmDrive);
if (dwRes != NO_ERROR) {
    /* place error handling code here */
    exit(-1);
}

if (parmDrive.FeaturesLow & TAPE_DRIVE_ERASE_LONG)
    printf("Device supports the long erase technique.\n");

...
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

Possible Serial Baud Rates on Various Machines

PSS ID Number: Q99026

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

Computers running Windows NT may be unable to set the same serial baud rates due to differences in serial port hardware on various platforms and machines. These differences may be important to note when writing a serial communications application that runs on different Windows NT platforms.

The simplest way to determine what baud rates are available on a particular serial port is to call the GetCommProperties() application programming interface (API) and examine the COMMPROP.dwSettableBaud bitmask to determine what baud rates are supported on that serial port.

MORE INFORMATION

================

Some baud rates may be available on one machine and not on another because of differences in the serial port hardware used on the two machines. Most Intel 80x86 machines use a standard 1.8432 megahertz (MHz) clock speed on serial port hardware, and therefore most Intel machines can set the same baud rates. However, on other platforms, such as MIPS, there is no standard serial port clock speed. MIPS serial ports are known to exist with 1.8432 MHz, 3.072 MHz, 4.2336 MHz, and 8.0 MHz serial port clock chips. Future NT implementations on other platforms may have different serial port clock speeds as well.

Furthermore, certain requested baud rates are special-cased in the Windows NT serial driver so that they will work. The following are these special cases:

MHz	Requested Baud	Divisor	Resulting Baud Rate (+/- 1)
1.8432	56000	2	57600
3.072	14400	13	14769
4.2336	9600	28	9450
4.2336	14400	18	14700
4.2336	19200	14	18900
4.2336	38400	7	37800
4.2336	56000	5	52920
8.0	14400	35	14286
8.0	56000	9	55556

The actual baud rate can be calculated by dividing the divisor multiplied

by 16 into the clock rate. For example, for a 1.8432 MHz clock and a divisor of 2, the baud rate would be:

```
1843200 Hz / (2 * 16) = 57600
```

For all other cases, as long as the requested baud rate is within 1 percent of the nearest baud rate that can be found with an integer divisor, the baud rate request will succeed.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseCommapi

Posting Frequent Messages Within an Application

PSS ID Number: Q40669

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The object-oriented nature of Windows programming can create a situation in which an application posts a message to itself. When such an application is designed, care must be taken to avoid posting messages so frequently that system messages to the application are not processed. This article discusses two methods of using the PeekMessage() function to combat this situation.

MORE INFORMATION

In the first method, a PeekMessage() loop is used to check for system messages to the application. If none are pending, the SendMessage() function is used from within the PeekMessage() loop to send a message to the appropriate window. The following code demonstrates this technique:

```
while (fProcessing)
     {
     if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
          {
          if (msg.message == WM_QUIT)
               break;
          /* process system messages */
          }
     else
          {
          /* perform other processing */
          ...
          /* send WM_USER message to window procedure */
          SendMessage(hWnd, WM_USER, wParam, lParam);
          }
     }
```

In the second method, two PeekMessage() loops are used, one to look for system messages and one to look for application messages. PostMessage() can be used from anywhere in the application to send the messages to the appropriate window. The following code demonstrates this technique:

```
while (fProcessing)
```

```
     {
     if (PeekMessage(&msg, NULL, 0, WM_USER-1, PM_REMOVE))
          {
          if (msg.message == WM_QUIT)
               break;
          /* process system messages */
          }
     else if (PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_REMOVE))
          /* process application messages */
     }
```

An application should use a PeekMessage() loop for as little time as possible. To be compatible with battery-powered computers and to optimize system performance, every Windows-based application should inform Windows that it is idle as soon and as often as possible. An application is idle when the GetMessage() or WaitMessage() function is called and no messages are waiting in the application's message queue.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMsg

PRB: "Help Author On" Appears in Help Title Bar

PSS ID Number: Q147864

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS
========

After you run Microsoft Help Workshop, the text in the title bar for all of your help files may display a number followed by the following text:

   (Help Author On)

This text will replace the title of the help file that normally appears.

CAUSE
=====

The "Help Author" command is provided by the Help Workshop to provide additional authoring information and debugging capabilities. Whenever Help Author is active, the title bar will display the topic number of the currently displayed topic and the "Help Author On" message.

By design, once turned on, Help Author remains active until it is disabled, even if the Help Workshop is no longer running.

RESOLUTION
==========

To disable Help Author, clear the Help Author check box in the Help Workshop's File menu. For the change to take effect, you must exit the Help Workshop.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

Under Windows 95 or Win32s, this setting corresponds to the following line in the [Windows Help] section of the Win.ini file. Note that a value of 1 indicates that Help Author is enabled and a value of 0 indicates that Help Author is disabled:

   Help Author=1

Under Windows NT, this setting corresponds to the following entry in the registry:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows Help\
Help Author:REG_SZ:1
```

Additional reference words: 1.30 4.00 hcw compiler
KBCategory: kbtool kbprb
KBSubcategory: TlsHlp

PRB: "Out of Memory Error" in the Win32 SDK Setup Sample

PSS ID Number: Q114610

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

When a dialog box is shown using UIStartDlg() or a billboard is switched during the file copy operation, you may receive an "out of memory" error. The error will also occur in any setup program based on a modified version of the SDK sample.

CAUSE

The dialog box and billboard templates are stored as resources in MSCUISTF.DLL. This DLL (Dynamic Link Library) is not loaded at the beginning of the setup program but is rather loaded and unloaded [using LoadLibrary() and FreeLibrary()] around each call to UIStratDlg() and when billboards are switched. Hence, each time a dialog or billboard is displayed, floppy disk #1 has the potential of being accessed. If disks have been swapped due to the installation process such that disk #1 is no longer in the drive, you will receive an out of memory error when LoadLibrary() is called on MSCUISTF.DLL.

RESOLUTION

To solve the problem, call LoadLibrary() at the beginning of WinMain() and call FreeLibrary() and the end of WinMain(). This way the DLL is always in use and will never be unloaded until the setup is done.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbprb
KBSubcategory: TlsMss

PRB: "Routine Not Found" Errors in Windows Help

PSS ID Number: Q108722

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

SYMPTOMS
========

When Freecell is running on Win32s, you may get some errors when you try to use Freecell Help. This has nothing to do with whether or not Win32s is installed correctly.

When you first open Help in Freecell running on Win32s, you will get a message box that says, "Routine Not Found." You will get that same message box when you choose the Find button.

When you choose any term in Freecell Help that has a dotted underline (a pop-up hot key), a window pops up but then quickly disappears. You will then get a message box that reads, "Help Topic Does Not Exist."

CAUSE
=====

FREECELL.HLP was meant to be used with WINHLP32.EXE. FREECELL.HLP uses the advanced features of WINHLP32.EXE for full-text searching. WINHELP.EXE, which runs on Windows 3.1, does not support full-text searching. WINHLP32.EXE does not run on Win32s or Windows 3.1. As a result, everytime FREECELL.HLP tries to bind the Find button to the full-text searching functions, it fails, and WinHelp pops up an error message box.

RESOLUTION
==========

You can still read the information in the help file. Although you cannot use the Find button to do full-text searches, you can use the Search button to do keyword searches. WINHLP32.EXE will not work on a MS-DOS + Windows + Win32s operating system. If you need to read the pop-up definitions or use the Find button, you can do so if you run Freecell Help on the Windows NT operating system.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: 16-bit Module Name Not Included in Toolhelp32 Snapshots

PSS ID Number: Q137288

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

The snapshots returned by Toolhelp32 identify 16-bit processes as the KERNEL32 module. In other words, the szExePath field of the MODULEENTRY32 structure contains KERNEL32.DLL for 16-bit Windows-based applications. The correct module is given for Win32-based applications.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

16-bit modules are very different from 32-bit modules and code that manipulates the internals of one kind usually must be rewritten for the other kind. The most common exception is code that prints out the name of the .exe file. This information can be obtained, for both 16-bit and 32-bit applications, from the szExeFile member of the PROCESSENTRY32 structure.

NOTE: The szExeFile field was added after the first beta of Windows 95, so some prerelease SDK header files do not include this field. It is important that you use the latest SDK header files for development.

REFERENCES
==========

The PVIEW95 SDK sample uses the module fields in the MODULEENTRY32 and PROCESSENTRY32 structures.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbprb
KBSubcategory:

PRB: _getdcwd() Returns the Root Directory Under Win32s

PSS ID Number: Q98286

The information in this article applies to:

 - Microsoft Win32s versions 1.1 and 1.2

SYMPTOMS
========

In the following code segment, _getdcwd() always returns the root:

```
   _getdcwd( 3, cBuf, MAX_PATH );
   MessageBox( hWnd, cBuf, "Drive 3 <C drive>", MB_OK );
```

Also, in the following code segment, _chdrive() and _getcwd() always return the root:

```
   _chdrive( 3 );
   _getcwd( cBuf, MAX_PATH );
   MessageBox( hWnd, cbuf, "Drive 3 <C drive>", MB_OK );
```

CAUSE
=====

When a Win32-based application starts on Win32s, the root is set as the current directory for any drive except the default drive.

RESOLUTION
==========

The following code fragments work as expected under Win32s:

```
   _getdcwd( 0, cBuf, MAX_PATH );
   MessageBox( hWnd, cBuf, "Drive 0 <default drive>", MB_OK );

   -or-

   GetCurrentDirectory( sizeof (cBuf), cBuf );
   MessageBox( hWnd, cBuf, "SCD", MB_OK );
```

MORE INFORMATION
================

Windows NT uses the current directory of a process as the initial current directory for the current drive of a child process. So for example, if the current directory in the command prompt (CMD.EXE) is C:\WINNT then the current directory of the child process will be C:\WINNT.

However, on Win32s, the current directory for any drive except the default drive is set to the root and not the current directory of the parent process. A Win32-based application running on Win32s calling _chdrive() or SetCurrentDirectory() to change the drive GetCurrentDirectory or _getcwd()

will then return the root. The _getdcwd() function is a composite of changing drives, getting the current directory of that drive, and change back to the original drive. Therefore, _getdcwd() will always return the root on Win32s.

Running the following sample to display the current directory of drives C and D under Windows NT properly displays the full path of the drive. Running the sample under Win32s always displays the root ("C:\", "D:\").

Sample Code

```
#include <direct.h>

...

   status =  _getdcwd(3, szPath, MAX_PATH);    // drive 3 == C:
   if( status != NULL )
   {
      MessageBox( hWnd, szPath, "Current working directory on C:", MB_OK );
   }

   status =  _getdcwd( 4, szPath, MAX_PATH );  // drive 4 == D:
   if( status != NULL )
   {
      MessageBox( hWnd, szPath, "Current working directory on D:", MB_OK );
   }

...
```

Additional reference words: 1.10 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: Access Denied When Opening a Named Pipe from a Service

PSS ID Number: Q126645

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51

SYMPTOMS
========

If a service running in the Local System account attempts to open a named pipe on a computer running Windows NT version 3.5, the operation may fail with an Access Denied error (error 5). This can happen even if the pipe was created with a NULL DACL.

NOTE: For more information about placing a NULL DACL on a named pipe, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q102798
   TITLE   : Security Attributes on Named Pipes

CAUSE
=====

In Windows NT version 3.1, a process running in the Local System account could connect to a resource using a Null Session. For security reasons, use of the Null Session is restricted by default on Windows NT version 3.5.

RESOLUTION
==========

You can allow access to a named pipe using the Null Session by adding the pipe name to the following registry entry on the machine that creates the named pipe:

   \HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\LanmanServer\
      Parameters\NullSessionPipes

The pipe name added to this entry is the name after the last backslash in the string used to open the pipe. For example, if you use the following string to open the pipe:

   \\hardknox\pipe\mypipe

you would add mypipe to the NullSessionPipes entry on the computer named hardknox.

You must either reboot or restart (stop and then start) the Server service for changes in this entry to take effect. Also, the named pipe will still need to have a NULL DACL.

In Windows NT 3.51, by customer request, it is no longer necessary to reboot. Once a named pipe is added to the key listed above, null-session connections to that pipe will immediately be accessible.

This new functionality allows programs to permit null session access to named pipes that do not have names known prior to booting the system.

MORE INFORMATION

Usually, when a session is established between a computer supplying a resource (server) and a computer that wants to use the resource (client), the client is identified and credentials are verified. When a Null Session is used, there is no validation of the client; everyone is allowed access.

If you allow a pipe to be used by a Null Session, you should either:

- Verify that the data supplied by the pipe is truly public.

  -or-

- Use an alternative method for verifying clients.

REFERENCES

The "Windows NT Registry Entries" help file in the Windows NT version 3.5 Resource Kit.

Additional reference words: 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseSecurity

PRB: AccessCheck() Returns ERROR_INVALID_SECURITY_DESCR

PSS ID Number: Q115946

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS
========

In certain cases, the AccessCheck() API fails and GetLastError() returns the message "ERROR_INVALID_SECURITY_DESCR". This error message indicates that the security descriptor passed to AccessCheck() was in an invalid format.

CAUSE
=====

This is expected behavior for the AccessCheck() function. AccessCheck() was designed for use by programs that create and maintain their own security descriptors. These security descriptors would always have the owner, DACL, and group information.

RESOLUTION
==========

If the security descriptor is indeed valid, you can eliminate the error by ensuring that the security descriptor has been opened for access to the following types of security information:

```
OWNER_SECURITY_INFORMATION
GROUP_SECURITY_INFORMATION
DACL_SECURITY_INFORMATION
```

You can double check the validity of the security descriptor by calling the IsValidSecurityDescriptor() API.

Additional reference words: 3.10 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseSecurity

PRB: After CreateService() with UNC Name, Service Start Fails

PSS ID Number: Q127862

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SYMPTOMS
========

When giving a Universal Naming Convention (UNC) name to CreateService(), the call succeeds, but service start fails with ERROR_ACCESSS_DENIED (5). The access denied message is given, even though the service runs under an account that has logon as service rights and has full access to the service image path.

CAUSE
=====

The service control manager calls CreateProcess() to start the service and the service controller runs in the LocalSystem security context. When you call CreateProcess() with a UNC name from a process running in the LocalSystem context, you get ERROR_ACCESS_DENIED. This is because LocalSystem has restricted (less than guest) access to remote machines. A null session is set up for LocalSystem remote access, which has reduced rights (less in Windows NT version 3.5 than in Windows NT version 3.1).

RESOLUTION
==========

There are two possible solutions:

- When specifying the fully qualified path to the service binary file, do not use a UNC name. It may be desirable to copy the service binary file to the local machine. This has the added benefit that the service will no longer be dependent on network operations.

  -or-

- If the service binary is on \\MACHINEA\SHARENAME, add SHARENAME to

  HKEY_LOCAL_MACHINE\SYSTEM\
  CurrentControlSet\
  Services\
  LanmanServer\
  Parameters\
  NullSessionShares

  on MACHINEA. This will let requests to access this share from null sessions succeed.

WARNING: This will allow everyone access to the share. If you want to maintain security for the share, create an account with the access required.

Additional reference words: 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseService

PRB: Antialiased Polygons Not Drawn in OpenGL Antipoly Sample

PSS ID Number: Q139653

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

The antipoly sample in OpenGL SDK BOOK directory is unable to draw antialised polygons with the generic implementation of Windows NT and Windows 95 OpenGL.

CAUSE

The technique used in this sample uses alpha values rather than depth values to determine what to write to the color buffer. You need to first turn off the depth buffer, choose the (GL_SRC_ALPHA_SATURATE, GL_ONE) blend function, and then draw polygons from front to back. But the generic Windows NT and Windows 95 OpenGL implementation does not provide alpha buffer. The cAlphaBits field of the pixel format descriptor for all generic pixel formats is 0.

RESOLUTION

The sample will work if an OpenGL accelerator card is present and additional device pixel formats provided by the OpenGL accelerator card support alpha buffer.

STATUS

This behavior is by design.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbprb
KBSubcategory: GdiOpenGL

PRB: App Desktop Toolbars Must Have WS_EX_TOOLWINDOW Style

PSS ID Number: Q132965

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SYMPTOMS

A problem can occur when an Application Desktop Toolbar (AppBar) is registered and the original position is set. The system reserves the desktop space the AppBar requested by moving all other shell objects and windows out of that rectangle. The problem is that the system then moves the AppBar window out of that rectangle as well.

CAUSE

When an AppBar requests some area on the desktop to occupy, the shell enumerates the windows in the task list and moves those windows outside of the newly reserved area. If your AppBar window appears in the task list it too is moved outside of the area.

RESOLUTION

When creating an Application Desktop Toolbar, the AppBar window must have the WS_EX_TOOLWINDOW window style for the shell to handle the AppBar window correctly. The WS_EX_TOOLWINDOW style prevents the AppBar window from appearing in the task list. The WS_EX_PALETTEWINDOW style can also be used because it includes the WS_EX_TOOLWINDOW style.

STATUS

This behavior is by design.

Additional reference words: AppBar 4.00 Windows 95
KBCategory: kbui kbprb
KBSubcategory: UsrShell

PRB: Applications Cannot Change the Desktop Bitmap

PSS ID Number: Q74366

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

The desktop bitmap is not updated when an application updates the Wallpaper entry of the [Desktop] section of WIN.INI and then sends a WM_WININICHANGE message to the desktop window.

RESOLUTION

By design, there is no supported method for an application to dynamically change the desktop bitmap under Windows 3.0 and 3.1.

MORE INFORMATION

Please note that an application could accidentally (or maliciously) reference a desktop bitmap in a format that would GP fault the system. For additional information, please see the following article(s) in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q69292
TITLE     : PRB: Video Driver GP Faults When Handling Large Bitmaps
```

Because the entry in WIN.INI has changed, this means that Windows will GP fault every time the user tries to start it in the future, making Windows no longer available.

In Windows 3.1, the application can call

```
SystemParametersInfo(SPI_SETDESKWALLPAPER,....)
```

which has safety checks built in.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 WM_WININICHANGE
KBCategory: kbui kbprb
KBSubCategory: UsrIni

PRB: Area Around Text and Remainder of Window Different Colors

PSS ID Number: Q22242

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When text is painted into a window, an area around the text is a different color than the remainder of the window.

CAUSE

The area around the text is painted with a solid color while the remainder of the window is painted using a dithered color.

RESOLUTION

To make the area around the text and the remainder of the window have the same color, perform one of the following two steps:

- Use a solid color for the window background, and use the same color for the text background. To ensure that a color is a solid color, use the GetNearestColor function. This function returns the nearest solid color available to represent the specified color.

  -or-

- Call the SetBkMode function to specify TRANSPARENT mode for the text. Doing so prevents Windows from painting the area behind the text. The window background color shows through instead.

MORE INFORMATION

By default, when an application paints text into its window, Windows fills the area around the character with the current background color. Windows always uses a solid color for this purpose.

When an application registers a window class, it specifies a handle to a brush that Windows uses to paint the window background. On some output devices, brushes can create dithered colors. On one of these

devices, the area behind painted text might have a different color
than the remainder of the window.

The following code specifies the window background color:

```
#define ELANGREEN  0x003FFF00
pTemplateClass->hbrBackground = CreateSolidBrush((DWORD)ELANGREEN);
```

The following code specifies the color used to paint around text and
draws some text into a device context:

```
#define SZ -1
SetBkColor(hDC, (DWORD)ELANGREEN);
DrawText(hDC, (LPSTR)"text", SZ, (LPRECT)&Rect, DT_BOTTOM);
```

The color used to paint the area around the text has a yellow cast,
which gives it a slightly different appearance than the window
background color.

A brush may be able to represent a wider color range than the solid
colors because a brush covers an area while a solid color may be used
to paint nominal-width lines (for example, lines that are one device
unit wide) that must be the same color at all locations and angles.
Therefore, the device-driver writer has the option of providing
dithered colors for brushes, but has no such freedom when it comes to
the solid colors for drawing lines.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic kbprb
KBSubcategory: GdiDrw

PRB: Average & Maximum Char Widths Different for TT Fixed Font

PSS ID Number: Q92410

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

The tmAveCharWidth and tmMaxCharWidth fields of the TEXTMETRIC structure are not equal for a fixed-pitch TrueType (TT) font such as Courier New.

RESOLUTION

This is by TrueType design. The tmMaxCharWidth denotes the maximum possible ink width of the font rather than maximum cell width of the font.

MORE INFORMATION

TrueType fonts use ABC character spacing. The "A" width is the distance that is added to the current position before placing the TrueType glyph. The "B" width is the width of the black part (ink width) of the TT glyph. The "C" width is the distance from the end of the "B" width to the beginning of the next character. Advanced width (cell width) is equal to A+B+C.

The physical TT fonts that are passed to drivers have just the "B" part of the characters, so all fonts at the level of the driver appear to be proportional width fonts. The tmMaxCharWidth is the least width in which the "B" part of all characters will fit. The tmAveCharWidth is the average advance width (A+B+C). For a fixed-pitch TT font such as Courier New, the A+B+C width is the same for all characters; however, the maximum width as defined above can be different.

tmMaxCharWidth is greater than tmAveCharWidth only if A+C is negative. This is possible for a fixed-pitch font.

Please see section 18.2.4.1 of the Windows 3.1 SDK "Guide to Programming" for more information about ABC character spacing.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic kbprb

KBSubcategory: GdiTt

PRB: Bad Colors or GPF Using Logical Palettes w/ DIB Sections

PSS ID Number: Q137232

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

SYMPTOMS
========

The program displays the wrong colors or causes a general protection (GP) fault when using a DIB section and a custom palette in a memory DC.

CAUSE
=====

There is a problem in Windows 95 where an internal palette structure is not set up until the palette is selected into a screen DC.

RESOLUTION
==========

If you use logical palettes with bitmaps created by CreateDIBSection, select the palette into a screen DC when the palette is created. Then the palette can be used normally with Screen DCs as well as with DIB sections.

STATUS
======

Microsoft is researching this behavior and will post new information here in the Microsoft Knowledge Base when it becomes available.

Additional reference words: 4.00 Windows 95 SelectObject SelectPalette CreatePalette CreateCompatibleDC GPF
KBCategory: kbgraphic kbprb
KBSubcategory: GdiPal

PRB: Bad Results Using Both ReadConsole and ReadConsoleInput

PSS ID Number: Q137195

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

Unpredictable results occur if you use ReadConsole and ReadConsoleInput together in a Windows 95-based application.

RESOLUTION
==========

Do not mix the two functions when using a single console.

STATUS
======

This behavior is by design.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbprb
KBSubcategory: BseCon

PRB: Bitmap Displays Upside Down Using StretchDIBits

PSS ID Number: Q151920

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:
  - Windows NT, version 3.51
  - Windows 95

SYMPTOMS

Calling StretchDIBits on a DIB causes it to display upside down.

CAUSE

Two situations could cause a bitmap to display upside down in Win32:

1. The signs of the cxSrc and cxDest parameters differ. The documentation for StretchDIBits states that the function creates a mirror image of a bitmap along the x-axis if the signs of the cxSrc and cxDest parameters differ. Whether intentional or unintentional, this could cause a bitmap to display upside down.

2. The sign of the biHeight value in the BITMAPINFOHEADER does not correctly reflect how the bitmap bits are stored in the DIB file. In Win16, all DIBs are stored bottom-up, with the bottom-most scan line stored first in the DIB file. In Win32, DIBs may also be stored top-down, with the top-most scan line stored first. Top-down DIBs are denoted by a negative biHeight value in the BITMAPINFOHEADER structure; bottom-up DIBS are denoted by a positive biHeight value:

DIB stored as	Should have a biHeight value of
top-down	negative
bottom-up	positive

Make sure the biHeight value is appropriately set as positive or negative, depending on how the bitmap bits were stored in the DIB file. A mismatch in the way this value is set could result to an upside down bitmap because the DIB engine will interpret the bits incorrectly. For example, a positive biHeight value for a DIB stored as top-down should cause a bitmap to display upside down.

STATUS

This behavior is by design.

Additional reference words: 4.00 3.51 inverted reverse
KBCategory: kbgraphic kbprb
KBSubcategory: GdiBmp

PRB: BSTR Redefinition If SHLOBJ.H Included in MFC Apps

PSS ID Number: Q132966

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SYMPTOMS

When you include SHLOBJ.H in an MFC application, the following compiler error occurs:

```
error C2371: 'BSTR' : redefinition
```

CAUSE

The SDK header file OAIDL.H and the MFC library both define a type BSTR.

RESOLUTION

To work around this problem, add the following definition before including any MFC headers. In an AppWizard-generated application this can be done at the beginning of the STDAFX.H include file:

```
#define _AFX_NO_BSTR_SUPPORT
```

Additional reference words: MFC BSTR 4.00 Windows 95
KBCategory: kbui kbprb
KBSubcategory: UsrShell

PRB: Building SDK SNMP Samples Results in Unresolved Externals

PSS ID Number: Q129240

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SYMPTOMS

Building the SNMP samples in the Win32 SDK for Windows NT version 3.5 (TESTDLL.DLL or SNMPUTIL.EXE) using Visual C++ version 2.0 results in the the linker complaining of unresolved externals _mb_cur_max_dll and _pctype_dll.

CAUSE

The application was built to use the C run-time in a DLL. In the Win32 SDK, the import library is CRTDLL.LIB, and in Visual C++, the import library is MSVCRT.LIB. The "___mb_cur_max_dll" and "__pctype_dll" symbols are defined in the CRTDLL.LIB file, but not in the MSVCRT.LIB file.

For more information, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119503
TITLE : PRB: Unresolved ___mb_cur_max_dll and __pctype_dll

RESOLUTION

Choose Settings from the Project menu. Then select C/C++, and go to the Code Generation category. For the run-time library listed, use Multithreaded using DLL.

Additional reference words: 3.50
KBCategory: kbnetwork kberrmsg kbprb
KBSubcategory: NtwkSnmp

PRB: Byte-Range File Locking Deadlock Condition

PSS ID Number: Q117223

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS

It is possible for a thread in a multithreaded Win32-based application to block while doing a LockFile() or LockFileEx() API call, even when the area of memory the thread has requested is not locked by another thread.

NOTE: If performance-monitoring tools (such as PERFMON) are used to examine the status of existing threads and the Thread State indicates that the thread is waiting and the Thread Wait Reason shows that it is the executive that the thread is waiting on, this is probably not an indication that a deadlock has occurred, because threads are often in this state for other reasons. Also, if the Thread State indicates that the thread is not waiting, then a deadlock has probably not occurred.

CAUSE

There is a small window of time during which a multithreaded application is vulnerable to this condition. Specifically, if one thread (call it Thread1) is in the process of unlocking a currently locked byte range within a file while a second thread (Thread2) is in the process of obtaining a lock on that same byte range using the same file handle and without specifying the flag LOCKFILE_FAIL_IMMEDIATELY, Thread1 can block, waiting for the region to become available. Ordinarily, when unlocking takes place, blocked threads are released; but in this critical window of time, it is possible for Thread2 to unlock the byte range without Thread1 being released. Thus, Thread1 never resumes operation despite the fact that there is no apparent fault in the logic of the program.

RESOLUTION

The deadlock condition described above can only come about if multiple threads are concurrently doing synchronous I/O using the same file handle.

To avoid the problem, you have three options:

- Each thread can obtain its own handle to the file either through the use of the DuplicateHandle() API or through multiple calls to the CreateFile() API.

  -or-

- The threads can perform asynchronous I/O. This also requires the application developer to provide some form of explicit synchronization to coordinate access to the file by the threads.

  -or-

- The threads can specify LOCKFILE_FAIL_IMMEDIATELY and then loop until a retry succeeds if the lock request fails. This option is the least desirable because it incurs significant CPU use overhead.

REFERENCES
==========

For more information about threads, files, and file handles, see the following sections in the "Win32 SDK Programmer's Reference," volume 2, part 3, "System Services":

- Chapter 43.1.6, "Synchronizing Execution of Multiple Threads"

- Chapter 45.2.2, "Reading, Writing, and Locking Files"

- Chapter 48.3, "Kernel Objects"

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubCategory: BseMisc

PRB: C2371 BSTR Redefinition When VFW.H Included in MFC App

PSS ID Number: Q129953

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), version 3.5
- Microsoft Visual C++, 32-bit Edition, versions 2.0 and 2.1

SYMPTOMS

The following error message occurs when compiling an application with Microsoft Visual C++, version 2.0 or 2.1, which uses the Microsoft Video for Windows header file, VFW.H, and the Microsoft Foundation Classes (MFC):

```
c:\msvc20\include\oleauto.h(214) : error C2371: 'BSTR' : redefinition;
different basic types
```

This error message does not occur when using Microsoft Visual C++ version 1.5 for Windows with the Microsoft Video for Windows 1.1 DK installed while running under Microsoft Windows operating system
version 3.1.

RESOLUTION

If the application does not require binary string (BSTR) support you can eliminate this error message by defining "_AFX_NO_BSTR_SUPPORT" before the MFC include files. For example, place the code below at the beginning of the STDAFX.H file:

```
#define _AFX_NO_BSTR_SUPPORT
```

If the application does require BSTR support, then you can eliminate this error message by including the code below before including the VFW.H file:

```
#define OLE2ANSI
```

MORE INFORMATION

This error message occurs by design. MFC versions 3.0 and 3.1 require either that "OLE2ANSI" is defined when including the object linking and embedding (OLE) headers or that "_AFX_NO_BSTR_SUPPORT" is defined. You cannot use both ANSI-BSTRs (which is the default) and Unicode-BSTRs; you must use one or the other.

The AFX.H file defines the BSTR type in order to allow CStrings to support BSTRs. The VFW.H file unconditionally includes the OLE2.H file, which eventually includes the OLEAUTO.H file, which also defines the BSTR type. "_AFX_NO_BSTR_SUPPORT" disables CString support for BSTRs.

Steps to Reproduce Problem

---

1. Generate a default application with AppWizard.
2. Add the statement "#include vfw.h" to the end of the STDAFX.H file.
3. Compile the application.

The VFW.H file is included with the Microsoft Visual C++, 32-bit Edition, and the Microsoft Win32 SDK for Windows NT.

Additional reference words: 3.10 3.50 2.00 2.10 CString port porting VfWDK
KBCategory: kbmm kbprb kberrmsg
KBSubcategory: MMVIDEO

PRB: Calling LoadMenuIndirect() with Invalid Data Hangs System

PSS ID Number: Q131281

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

Under Windows 95 if you call LoadMenuIndirect() with invalid data, the system hangs (stops responding).

CAUSE
=====

Invalid data anywhere in the array of MENUITEMTEMPLATE structures passed to the LoadMenuIndirect() function may cause this problem. An example of this might be an extra NULL byte after a MENUITEMTEMPLATE structure.

Under Windows NT version 3.5, passing the same invalid data in the MENUITEMTEMPLATE structure causes LoadMenuIndirect() to return NULL, with GetLastError() reporting an ERROR_INVALID_DATA value.

STATUS
======

This behavior is by design.

Additional reference words: 4.00 freeze lock up Menus
KBCategory: kbui kbprb
KBSubcategory: UsrMen

PRB: Can't Center Explorer-Style FileOpen Common Dialog Box

PSS ID Number: Q140722

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95 and Windows NT 3.51

SYMPTOMS

Generally, an application that attempts to alter the default behavior of a dialog box before it even comes up on the screen, would process the WM_INITDIALOG message. For common dialogs, this message is processed in the common dialog's hook procedure. However, attempting to center the new Explorer-Style FileOpen Common Dialog in Windows 95 in the hook procedure's WM_INITDIALOG case will result in an off-center dialog box.

CAUSE

This problem arises with the new Explorer-Style common dialog only when the OFN_ENABLETEMPLATE flag is set in the OPENFILENAME structure, and a dialog template is specified in lpTemplateName that includes the new controls that you want to add to the dialog box.

Typically, the code to center a dialog box would first get the dimensions of the dialog box by using GetWindowRect so it could be centered on the screen appropriately. At WM_INITDIALOG time, the FileOpen dialog has not been resized to accomodate the new controls specified in the lpTemplateName member of the OPENFILENAME struct; therefore, GetWindowRect() returns the dimensions of the original FileOpen dialog, not that of the actual customized dialog that comes up on the screen. This then causes the dialog box to come up a few pixels off-center.

RESOLUTION

Call GetWindowRect() and the rest of the code to center the dialog box, in the hook procedure's WM_NOTIFY case, with code set to CDN_INITDONE, a new common dialog notification code for Windows 95. By this time, all the controls have been added in, and the dialog has been resized accordingly.

STATUS

This behavior is by design.

MORE INFORMATION

Here's how the Win32 SDK COMDLG32 sample can be modified to do this. Note that COMMDLG creates the dialog specified by lpTemplateName as a child of

the standard FileOpen common dialog box. As a result, the hDlg passed to the application's hook function will be the child of the standard FileOpen dialog box. To get a handle to the standard dialog box from the hook function, an application needs to call GetParent (hDlg).

Code Sample Modification of Win32 SDK COMDLG32 Sample

```
BOOL NEAR PASCAL TestNotify(HWND hDlg, LPOFNOTIFY pofn)
{
 switch (pofn->hdr.code)
 {

     case CDN_INITDONE:
     {
        CenterWindow (GetParent (hDlg), NULL);
        break;
     }

     :
     :
}

// center with respect to another window
// Specifying NULL for hwndParent centers hwndChild relative to the screen.
BOOL CenterWindow(HWND hwndChild, HWND hwndParent)
{
    RECT    rcChild, rcParent;
    int     cxChild, cyChild, cxParent, cyParent;
    int     cxScreen, cyScreen, xNew, yNew;
    HDC     hdc;

    // Get the Height and Width of the child window
    GetWindowRect(hwndChild, &rcChild);
    cxChild = rcChild.right - rcChild.left;
    cyChild = rcChild.bottom - rcChild.top;


    if (hwndParent)
    {
    // Get the Height and Width of the parent window
       GetWindowRect(hwndParent, &rcParent);
       cxParent = rcParent.right - rcParent.left;
       cyParent = rcParent.bottom - rcParent.top;
    }

    else
    {
       cxParent = GetSystemMetrics (SM_CXSCREEN);
       cyParent = GetSystemMetrics (SM_CYSCREEN);
       rcParent.left = 0;
       rcParent.top  = 0;
       rcParent.right = cxParent;
       rcParent.bottom= cyParent;
    }
```

```
    // Get the display limits
    hdc = GetDC(hwndChild);
    cxScreen = GetDeviceCaps(hdc, HORZRES);
    cyScreen = GetDeviceCaps(hdc, VERTRES);
    ReleaseDC(hwndChild, hdc);

    // Calculate new X position, then adjust for screen
    xNew = rcParent.left + ((cxParent - cxChild) / 2);
    if (xNew < 0)
    {
        xNew = 0;
    }
    else if ((xNew + cxChild) > cxScreen)
    {
        xNew = cxScreen - cxChild;
    }

    // Calculate new Y position, then adjust for screen
    yNew = rcParent.top  + ((cyParent - cyChild) / 2);
    if (yNew < 0)
    {
        yNew = 0;
    }
    else if ((yNew + cyChild) > cyScreen)
    {
        yNew = cyScreen - cyChild;
    }

    // Set it, and return
    return SetWindowPos(hwndChild,
                        NULL,
                        xNew, yNew,
                        0, 0,
                        SWP_NOSIZE | SWP_NOZORDER);
}
```

Additional reference words: 4.00
KBCategory: kbprg kbprb kbcode
KBSubcategory: UsrCmnDlg

PRB: Can't Disable CTRL+ESC and ALT+TAB Under Windows NT

PSS ID Number: Q125614

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.5

SYMPTOMS

CTRL+ESC and ALT+TAB task switching cannot be disabled by an application running under Windows NT.

Capturing WM_SYSCOMMAND messages and not sending them on for processing by DefWindowProc() allowed task switching to be disabled in Windows version 3.1, but it doesn't work under Windows NT.

CAUSE

A primary reason for this change is to avoid dependence on an application for processing of these key combinations. This way a hung application can be switched away from by using either CTRL+ESC or ALT+TAB.

RESOLUTION

CTRL+ESC may disabled on a system-wide basis by replacing the NT Task Manager. This is not recommended.

ALT+TAB and ALT+ESC may be disabled while a particular application is running if that application registers hotkeys for those combinations with Register HotKey().

STATUS

This behavior is by design.

REFERENCES

The first reference below describes the steps that must be taken to replace TASKMAN.EXE. The two additional references provide more information on the Windows NT Task Manager and its relationship to the Program Manager.

ARTICLE-ID:Q89373
TITLE   :Replacing the Windows NT Task Manager

ARTICLE-ID:Q100328
TITLE   :Replacing the Shell (Program Manager)

ARTICLE-ID:Q101659
TITLE :How Windows NT, 16-Bit Windows 3.1 Task Managers Differ

Additional reference words: 3.50
KBCategory: kbusage kbprb
KBSubcategory: UsrMisc

PRB: Can't Increase Process Priority

PSS ID Number: Q110853

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS

An attempt to increase process priority to the real-time priority REALTIME_PRIORITY_CLASS may not set the PriorityClass to the expected level.

CAUSE

Only accounts with the "Increase Scheduling Priority" permission can increase the priority to real-time. Only Administrators and "Power Users" have this permission by default.

RESOLUTION

Either run the program from an Administrator or Power User account, or grant the Increase Scheduling Priority permission to any user group that must run the program.

STATUS

This behavior is by design.

MORE INFORMATION

The Increase Scheduling Priority permission can be granted to a user or group through the User Manager. To do this:

1. Open the Administrative Tools group in Program Manager.
2. Run the User Manager application.
3. Choose User Rights from the Policies menu.
4. Select the Show Advanced User Rights check box.
5. Select Increase Scheduling Priority from the drop-down list box.
6. Choose the Add button to add users or groups to the list of entities that have this right.

Note that the call to SetPriorityClass() may return success even though the priority was not set to REALTIME_PRIORITY_CLASS, because if you don’t have the Increase Scheduling Priority permission, a request for REALTIME_PRIORITY_CLASS is interpreted as a request for the highest

priority class allowed in the current account. If it is important to know the actual priority class that was set, use the GetPriorityClass() function.

Additional reference words: 3.10 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseProcThrd

PRB: Can't Remove Minimize or Maximize Button from Caption Bar

PSS ID Number: Q130760

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SYMPTOMS

Under Windows 95, applications cannot create an overlapped or a popup window that contains only the Minimize or the Maximize button.

CAUSE

This is by design. Windows can have both buttons or none depending on the styles specified while creating the window, but specifying just one style (either the WS_MAXIMZIEBOX or WS_MINIMIZEBOX) creates both buttons on the caption, with the other one disabled.

STATUS

This behavior is by design.

MORE INFORMATION

Under Windows version 3.1 or Window NT, applications could remove either the Maximize or Minimize buttons on the caption bar. (This was usually done when the appliction removed a corresponding menu item from the system menu of a window.)

Applications running under Windows 95 that try to remove one of the buttons (not both), will not succeed. The system displays both buttons and disable the one whose style was not specified during creation. This is by design, and there is no way to work around it, unless the application draws its own caption bar.

Applications that removed the maximize or minimize menu items under Windows version 3.1, should just gray them out (disable them) under Windows 95 to maintain a uniform user interface.

Under Windows 95, applications can create a window (overlapped or popup) with just the Close button (the X button) by creating the window without specifying the WS_MAXIMIZEBOX or WS_MINIMIZEBOX styles. Calling SetWindowLong(GWL_STYLE) to change or remove the minimize of the maxmize buttons dynamically still displays both buttons with one of them disabled.

Additional reference words: 4.00 user

KBCategory: kbui kbprb
KBSubcategory: UsrWndw

PRB: Can't Use TAB to Move from Standard Controls to Custom

PSS ID Number: Q131283

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SYMPTOMS

As you attempt to customize the new Explorer-style File Open or Save As common dialog, pressing the TAB key fails to move the focus from the standard dialog's controls to the new controls specified in the custom template.

CAUSE

The custom dialog box template failed to specify the DS_CONTROL style.

RESOLUTION

When creating a custom dialog box template to be used with the new FileOpen or Save As common dialog, you must specify the DS_CONTROL style.

STATUS

This behavior is by design.

MORE INFORMATION

With the new way of customizing the File Open or Save As common dialog in Windows 95, you can customize applications to provide a custom dialog box template that adds controls to the default Open or Save As dialog box. This custom dialog box is created as a child of the default dialog box and thus requires a WS_CHILD style.

DS_CONTROL is a new style provided in Windows 95. It makes a modal dialog created as a child of another dialog interact well with its parent dialog. This style ensures that the user can move between the controls of the parent dialog and the child dialog - or in this case between the standard Open or Save As common dialog and the custom dialog provided by the application.

NOTE: This article does not apply to Windows NT. This information applies only to the new way of customizing the Explorer-style File Open or Save As common dialog in Windows 95.

Additional reference words: 4.00
KBCategory: kbui kbprb
KBSubcategory: UsrCmnDlg

PRB: Cannot Alter Messages with WH_KEYBOARD Hook

PSS ID Number: Q33690

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

After creating a program that uses the WH_KEYBOARD hook function to intercept a user's keystrokes and changing the wParam value before passing the hook code on to DefHookProc(), whichever application currently has the focus still receives the original character typed in rather than the translated character.

CAUSE

Keyboard messages cannot be altered with the WH_KEYBOARD hook. All that can be done is to "swallow" the message (return TRUE) or have the message passed on (FALSE). In a keyboard hook function, when you return DefHookProc(), you are passing the event to the next hook procedure in the potential hook chain, and giving it a chance to look at the event to decide whether or not to discard it. You are not passing the message to the system as if you had called DefWindowProc() from a Window procedure.

RESOLUTION

NOTE: In the discussion below, ignore the references to the WH_CALLWNDPROC hook for Win32-based applications. Win32 does not allow an application to change the message in a CallWndProc, as 16-bit Windows does.

To change the value of wParam (and hence the character message that is received by the window with the focus), you must install the WM_GETMESSAGE and WH_CALLWNDPROC hooks. The WH_GETMESSAGE hook traps all messages retrieved via GetMessage() or PeekMessage(). This is the way actual keyboard events are received: the message is placed in the queue by Windows and the application retrieves it via GetMessage() or PeekMessage(). However, because applications can send keyboard messages with SendMessage(), it is best to also install the WH_CALLWNDPROC hook. This hook traps messages sent to a window via SendMessage().

These hooks pass you the address of the message structure so you can change it. In Windows 3.0, when you return DefHookProc() within a WH_GETMESSAGE or WH_CALLWNDPROC hook procedure, you are passing the address of the (potentially altered) contents of the message structure on to the next hook

function in the chain. In Windows 3.1, you should use the CallNextHookEx() function to pass the hook information to the next hook function. If you alter the wParam before passing on the message, this will change the character message eventually received by the application with the focus.

NOTE: For Windows 3.0, keep in mind that the hook callback procedure must be placed in a DLL with fixed code so that it will be below the EMS line and thus will always be present. If the hook callback procedure is not in a fixed code segment, it could be banked out when it is called, and this would crash the system. System-wide hooks in Windows 3.1, however, must still reside in a DLL.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrHks

PRB: Cannot Change Icon at Runtime in Visual Basic 3.0 Apps

PSS ID Number: Q135142

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

You can't change the Icon property of a form at run time in a Visual Basic version 3.0 application.

CAUSE

Windows version 3.1 had code that looked at the first 100 icons in Visual Basic applications. Windows 95 code now looks only at the first five icons in Visual Basic applications. This change was made both for performance reasons and because later versions of Visual Basic (4.0 and beyond) no longer need this special code.

RESOLUTION

Update the Visual Basic version 3.0 application to a Visual Basic version 4.0 (or later) application.

STATUS

This behavior is by design.

Additional reference words: 95 development program 4.00
KBCategory: kbui kbprg kbinterop kbprb
KBSubcategory:

PRB: Cannot Change IME Status Setting on Windows NTJ or Win95J

PSS ID Number: Q152585

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  - Microsoft Windows NT, version 3.51
  - Microsoft Windows 95, version 4.0

SYMPTOMS

On the Japanese version of Windows NT 3.51 and Windows 95, using the MSIME95 as the Input Method Editor, if a process creates a second thread, the second thread, in turn, creates a window, while the primary thread's main window remains the foreground window. When this occurs, the user cannot change the IME status settings for the primary thread.

However, at this time, if a window of another thread is set as the foreground window by a mouse click to another window or by calling Windows API, then the window of the primary thread is reset as the main window, and the user will be able to change the IME status settings again by clicking the IME status window buttons.

RESOLUTION

To prevent this problem from occurring, immediately after the window of the second thread is created, use SetForegroundWindow() to set the window of the second thread as the foreground window very briefly, and then set the main window of the primary thread as the foreground window again, as follows:

```
SetForegroundWindow(hSecondThreadWnd);
SetForegroundWindow(hMainThreadWnd);
```

STATUS

This behavior is by design.

Additional reference words: 3.51 4.00 Input method editor FE
KBCategory: kbprg kbprb
KBSubcategory: wintldev

PRB: Cannot Load Module with Resource ID > 32767 (0x7FFF)

PSS ID Number: Q137248

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SYMPTOMS

The GetLastError() function returns error 11 ("An attempt was made to load a program with an incorrect format."), and the LoadLibrary(), WinExec() or CreateProcess() function fails if the module you are trying to load contains a resource identifier (ID) that is greater than 32767 (0x7FFF).

CAUSE

Microsoft Windows 95 uses the high bit in the 16-bit resource ID to determine if the resource ID is a numeric value or a string. When this bit is set, the operating system determines that the ID is a string. Therefore, it appears that the file is corrupted.

RESOLUTION

All applications must ensure that all of the resource IDs in their modules are less than 0x7FFF. These are resource IDs only (DIALOG, MENU, ICON, CURSOR, STRINGTABLE, BITMAP, ACCELERATOR, FONT, and so on); control and menu IDs are not included in this restriction.

Additional reference words: 1.30 4.00
KBCategory: kbui
KBSubcategory: UsrCtl W32s

PRB: CBS_SIMPLE ComboBox Repainting Problem

PSS ID Number: Q128110

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5
  - Microsoft Win32s version 1.2

SYMPTOMS

When a CBS_SIMPLE combo box has a WS_CLIPCHILDREN parent, the area below the edit control and left to the list box is not repainted correctly. This problem exists for 16-bit as well as 32-bit applications.

RESOLUTION

To work around this problem, subclass the combo box, calculate the blank area, and then repaint to the desired color.

The following ComboBox subclass procedure is written for a 16-bit application, but you can use the same idea in 32-bit applications.

Sample Code

```
LRESULT CALLBACK NewComboProc(
      HWND hWnd,
      UINT uMessage,
      WPARAM uParam,
      LPARAM lParam)
{
HDC myDC;
HPEN   hPen, hOldPen;
HBRUSH hBrush;
HBRUSH hOldBrush;
COLORREF myColor=RGB(255,255,255); //It can be any color. Here
               //the area is painted white.

HWND hEdit, hList;
RECT comboRect, editRect, listRect;
char   *wndClassName="Edit";

if (uMessage == WM_PAINT)
  {
  CallWindowProc(lpfnOldComboProc, hWnd, uMessage, uParam,
                 lParam);
  myDC = GetDC(hWnd);
```

```
  hBrush = CreateSolidBrush(myColor);
  hPen   = CreatePen (PS_SOLID, 1, myColor);
  hOldBrush = SelectObject(myDC, hBrush) ;
  hOldPen   = SelectObject(myDC, hPen);

  //This code obtains the handle to the edit control of the
  //combobox.

  hEdit = GetWindow(hWnd, GW_CHILD);
  GetClassName (hEdit, wndClassName, 10);
  if (!lstrcmp (wndClassName, "Edit"))

    hList=GetWindow(hEdit, GW_HWNDNEXT);

  else
    {
    hList=hEdit;
    hEdit=GetWindow(hList, GW_HWNDNEXT);
    }

  //The dimensions of the Edit Control, ListBox control and
  //the Combobox are  calculated and then used
  //as the base dimensions for the Rectangle() routine.

  GetClientRect (hWnd, &comboRect);
  GetClientRect (hEdit, &editRect);
  GetClientRect (hList, &listRect);
  Rectangle (myDC,
             comboRect.left,
             editRect.bottom,
             comboRect.right-listRect.right,
             comboRect.bottom);
  //Also paint the gap, if any exists, between the bottom
  //of the listbox and the bottom of the ComboBox rectangle.
  Rectangle (myDC,
             comboRect.right-listRect.right,
        editRect.bottom +
             listRect.bottom,
        comboRect.right,
        comboRect.bottom);

  DeleteObject(SelectObject(myDC, hOldBrush)) ;
  DeleteObject(SelectObject(myDC, hOldPen)) ;
  ReleaseDC(hWnd, myDC);
  return TRUE;
  }

return CallWindowProc(lpfnOldComboProc, hWnd, uMessage, uParam,
 lParam);
}
```

STATUS
======

This behavior is by design.

MORE INFORMAITON
================

Steps to Reproduce Behavior

To reproduce this behavior, use AppStudio to create a dialog with the WS_CLIPCHILDREN style, put a CBS_SIMPLE combobox in the dialog, and click the test button so you can test the dialog. Then move something on top of the dialog, and move the object on top of the combobox away. You can then see that area to the left of the listbox is not repainted correctly.

Additional reference words: 3.10 3.50 1.20
KBCategory: kbui kbcode kbprb
KBSubcategory: UsrCtl

PRB: CBT_CREATEWND Struct Returns Invalid Class Name

PSS ID Number: Q106079

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The CBT_CREATEWND structure contains information passed to a WH_CBT hook callback function before a window is created in the system. The lpszClass field of this CBT_CREATEWND structure may return an invalid class name, particularly for windows created internally by the system.

SYMPTOMS

The code to get to the window class name via the CBT_CREATEWND structure in a CBT callback function may look like the following:

```
   int FAR PASCAL CBTProc (int nCode, WPARAM wParam, LPARAM lParam)
    {
         LPCBT_CREATEWND  CreateWndParam;

           if (nCode >= 0)
           {
                switch (nCode)
                {
                   case HCBT_CREATWND:
                        CreateWndParam = (LPCBT_CREATEWND)lParam;
                        OutputDebugString ("ClassName = ");
                        OutputDebugString (lParam->lpcs->lpszClass);
                        OutputDebugString ("\r\n");
                        break;
                     :
                     :
                }
           }

           return (int)CallNextHookEx (hHook, nCode, wParam, lParam);
    }
```

However, this code may or may not output the correct class name, depending on what the call to CreateWindow() (for the window about to be created) looks like.

RESOLUTION

==========

Windows provides the GetClassName() function to allow applications to retrieve a window's class name. This function takes the handle to the window as a parameter, as well as a buffer to receive the null-terminated class name string. This function is a more reliable and a more recommended means to obtain window class name information than the CBT callback function's CBT_CREATEWND structure.

MORE INFORMATION
================

Whenever a window is about to be created, Windows checks to see if a WH_CBT hook is installed anywhere in the system. If it finds one, it calls the CBTProc() callback function with hook code set to HCBT_CREATEWND, and with the lParam parameter containing a long pointer to a CBT_CREATEWND structure.

The CBT_CREATEWND structure is defined in WINDOWS.H as follows

```
typedef struct tagCBT_CREATEWND {
    CREATESTRUCT FAR* lpcs;
    HWND hwndInsertAfter;
} CBT_CREATEWND;
```

with the CREATESTRUCT structure defined in the same file as:

```
typedef struct tagCREATESTRUCT {
   void FAR* lpCreateParams;
      :
   LPCSTR  lpszClass;    // Null-terminated string specifying window
                         // class name
   DWORD  dwExStyle;
} CREATESTRUCT;
```

When Windows internally creates windows (such as predefined controls in a dialog box, for example), it uses atoms for lpszClass instead of the actual string to minimize overhead. This makes it a little less straightforward (sometimes impossible) to get to the actual class name directly from the lpszClass field of the CREATESTRUCT structure.

To cite one particular example, when an application is minimized in Windows, Windows calls CreateWindow() for the icon title, specifying a class name of

```
(LPSTR)MAKEINTATOM (ICONTITLECLASS == 0x8004)
```

where MAKEINTATOM() is defined in WINDOWS.H as:

```
#define MAKEINTATOM (i)    ((LPCSTR) MAKELP (NULL, i))
```

Given this, to get to the actual class name, GetAtomName() must be called in this manner:

```
char szBuf [10];
```

```
   GetAtomName (LOWORD (lpszClass), szBuf, 10);
   OutputDebugString (szBuf);
```

This outputs a class name of #32772 for the IconTitleClass.

For predefined controls, such as "static", "button", and so forth, in a dialog box, the Dialog Manager calls CreateWindow() on each control, similarly using atoms for lpszClass. The Dialog Manager, however, creates these atoms in a local atom table in USER's default data segment (DS), thus making it impossible for other applications to get to these class names.

[Note the difference between local and global atom tables, where global atom tables are stored in a shared DS, and are therefore accessible to all applications, while local atom tables are stored in USER's default DS. DDE uses global atom tables so that Excel, for example, can use GlobalAddAtom() to add a string to the atom table, and another program could use GlobalGetAtomName() to obtain the character string for that atom.]

More information on atoms can be found by searching on the following word in the Microsoft Knowledge Base:

```
   atoms
```

More Information can also be found in the Windows version 3.1 online Help file under the heading Function Groups, under the Atom Functions subheading.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrHks

PRB: CFileDialog::DoModal() Does Not Display FileOpen Dialog

PSS ID Number: Q131225

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS

Calling CFileDialog::DoModal() returns without displaying the FileOpen common dialog.

CAUSE

The CFileDialog class will automatically use the new Explorer-style FileOpen common dialog under Windows 95. This can break existing code which customizes these dialogs with custom templates, because the mechanism has changed in Windows 95.

NOTE: This does not apply to Windows NT 3.51, as this version of Windows NT will not display the new Explorer-style dialog.

RESOLUTION

An application that depends on the old behavior of customizing the File Open common dialogs will need to reset the OFN_EXPLORER bit in the Flags member of the OPENFILENAME structure before calling CFileDialog::DoModal.

MORE INFORMATION

The DIRPKR sample in particular, exhibits the symptoms described above, and will need to be modified to display the dialog box correctly in Windows 95. It works as is under Windows NT 3.51.

Sample Code

```
    CMyFileDlg  cfdlg(FALSE, NULL, NULL, OFN_SHOWHELP | OFN_HIDEREADONLY |
                      OFN_OVERWRITEPROMPT | OFN_ENABLETEMPLATE,
                      NULL, m_pMainWnd);

    cfdlg.m_ofn.hInstance      = AfxGetInstanceHandle();
    cfdlg.m_ofn.lpTemplateName = MAKEINTRESOURCE(FILEOPENORD);
    cfdlg.m_ofn.Flags         &= ~OFN_EXPLORER;

    if (IDOK==cfdlg.DoModal())
    {
```

```
      :
      :
    }
```

NOTE: For versions of Visual C++ 2.x, the OFN_EXPLORER flag has not been defined. You can use this code instead:

```
    cfdlg.m_ofn.Flags &= ~(0x00080000);
```

REFERENCES
==========

This information was derived from Visual C++ 2.1 Technical Note 52: "Writing Windows 95 Applications with MFC 3.1"

Additional reference words: 2.10 4.00
KBCategory: kbui
KBSubcategory: UsrCmnDlg

PRB: COMM (TTY) Sample Does Not Work on Windows 95

PSS ID Number: Q128787

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS

The Win32 COMM (old TTY) sample that ships with Visual C++ version 2.x and the Windows 95 SDK (pre-release) does not work correctly under Windows 95 M8 builds and later. The problem involves assigning values to the Offset member of the OVERLAPPED structure which is one of the arguments to the WriteFile function call.

The observed behavior is that the COMM sample writes only one byte to the serial port. No other data is transmitted after the first byte.

CAUSE

The documentation for the OVERLAPPED structure explicitly states that the Offset and OffsetHigh members must be set to 0 when reading from or writing to a named pipe or communications device. This was not done in the sample.

RESOLUTION

1. Delete the following line from the WriteCommByte() function in the sample:

   ```
   WRITE_OS( npTTYInfo ).Offset += dwBytesWritten ;
   ```

2. Add the following lines to the CreateTTYInfo() function:

   ```
   WRITE_OS( npTTYInfo ).Offset = 0 ;
   WRITE_OS( npTTYInfo ).OffsetHigh = 0 ;
   ```

STATUS

This is a problem with the sample, not with Windows 95. Windows 95 correctly implements WriteFile() and use of the OVERLAPPED structure.

Additional reference words: 4.00
KBCategory: kbprg kbprb
KBSubcategory: BseComm

PRB: Connect() Fails, WSAGetLastError() returns WSAENETUNREACH

PSS ID Number: Q140014

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SYMPTOMS

A Win32 Winsock client issues a connect() call over SPX or SPXII protocols and the call fails with SOCKET_ERROR with WSAGetLastError() returning error WSAENETUNREACH(10051).

CAUSE

Either the IPX network number specified in the SOCKADDR_IPX address structure for the server is incorrect (network unreachable), or the server is not listening on the specified socket number.

STATUS

This behavior is by design.

MORE INFORMATION

If a Win32 Winsock clients issues a connect() call over TCP/IP protocol for a server and the server is not listening on the specified port, connect() returns SOCKET_ERROR and WSAGetLastError() returns error WSAECONNREFUSED(10061). However, the IPX/SPX implementation of Windows sockets on the Windows NT platform returns error WSAENETUNREACH for the same scenario.

Additional reference words: 3.51
KBCategory: kbprg kbnetwork kbprb
KBSubcategory: NtwkWinsock

PRB: Context Help Not Compatible with Min or Max Button

PSS ID Number: Q135787

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS

In Windows 95, the WS_EX_CONTEXTHELP extended window style is not compatible with the WS_MINIMIZEBOX or WS_MAXIMIZEBOX styles. The context help icon isn't displayed on the caption as long as one or both of the minimize or maximize boxes is displayed. The same thing happens with dialog boxes using the DS_CONTEXTHELP and the WS_MINIMIZEBOX or WS_MAXIMIZEBOX styles.

Note that the window must have the WS_SYSMENU style.

STATUS

This behaviour is by design.

Additional reference words: 4.00 1.30
KBCategory: kbui kbprb
KBSubcategory: UsrWndw

PRB: Controls Do Not Receive Expected Messages

PSS ID Number: Q121094

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, and 1.2

SYMPTOMS

A Win32-based application works as expected under Windows NT, but under Win32s, control messages are not received by the application as expected.

Here are some possible scenarios symptomatic of this problem:

- Sending LB_GETSELITEMS to a superclassed list box does not work.
- The custom control procedure does not receive messages such as CB_ADDSTRING, CB_INSERTSTRING, or CB_SHOWDROPDOWN.
- The wrong control message is received.

CAUSE

This is a Win32s limitation. When a Win32-based application sends a message, Win32s processes the message and passes it to Windows. Windows is responsible for delivering the message. Win32s truncates wParam from a DWORD to a WORD. Most messages do not use the high word of wParam, however, if you do use the high word of wParam, be aware that it will be lost under Win32s.

In the course of processing known messages (messages that are not user-defined), Win32s translates any pointers in lParam. In addition, if a message is new to Win32, the corresponding Windows message is used instead. For example, the Windows WM_CTLCOLOR was replaced in Win32 with WM_CTLCOLORBTN, WM_CTRLCOLORDLG, WM_CTLCOLOREDIT, and so on. Therefore, if a Win32-based application uses WM_CTRLCOLORBTN, Win32s passes WM_CTLCOLOR to Windows with a type of CTLCOLOR_BTN.

Control messages are not unique on Windows version 3.1 as they are on Windows NT. Control messages on Windows have values above WM_USER, however, messages for one control may share the same number as a message of another control. For example, both CB_ADDSTRING and LB_DELETESTRING are defined as WM_USER+3. Therefore, when Win32s receives the WM_USER+3 message, it needs to determine the correct control message. Win32s looks at the window class of the window that will receive the message and maps the message accordingly. If the window class is not a recognized control class, as in the case of a superclassed control, the message is not mapped, which results in unexpected behavior.

RESOLUTION

In order to get the desired behavior under Win32s, make the custom control use a recognized control class (such as "combobox") and subclass the window procedure instead of superclassing. If you need to subclass before the WM_CREATE/WM_NCCREATE messages, use a CBT hook. You will not be able to change the class icon and cursor, but the messages will be handled correctly.

If you need to use a custom control, create and use user-defined messages instead of the control messages.

STATUS
======

This behavior is by design.

Additional reference words: 1.10 1.15 1.20
KBCategory: kbprg kbprb
KBSubCategory: W32s

PRB: CreateDialogIndirect() Fails Under Windows 95

PSS ID Number: Q137618

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 4.0

SYMPTOMS

Programs that use CreateDialogIndirect() to create their dialog boxes at run time in Windows 95 may have trouble if the program has been ported from Windows NT or Windows 3.1.

CAUSE

The dialog resource for Windows 95 is the same as Windows NT; that is, all strings must be in Unicode, and all structures must be DWORD aligned.

Programs ported from Windows NT may find that when they insert strings into the dialog template resource, they used lstrcpyW() to force the string to be Unicode or the program itself was called compiled for Unicode. However, lstrcpyW() is not implemented in Windows 95, so it returns ERROR_NOT_IMPLEMENTED. To generate Unicode strings in Windows 95, the program must use MultiByteToWideChar().

Programs ported from 16-bit Windows 3.1 will probably have problems with the Unicode strings mentioned above and also with the structure alignment. In 16-bit Windows, dialog box resource structures are aligned on byte boundaries, so there's no work for the programmer to do. However, in Win32, all structures must be aligned on DWORD (four-byte) boundaries. AlignPtr() is a simple helper routine that takes a pointer and returns the nearest pointer aligned on a DWORD boundary. The program should call this between all the dialog resource structures.

RESOLUTION

Use MultiByteToWideChar() to generate Unicode strings in Windows 95, and call AlignPtr(), which takes a pointer and returns the nearest pointer aligned on a DWORD boundary, between all the dialog resource structures to align all structures on DWORD boundries.

STATUS

This behavior is by design.

MORE INFORMATION

```
//
// AlignPtr - Helper routine.  Take an input pointer, return closest
// pointer that is aligned on a DWORD (4 byte) boundary.
//

LPWORD AlignPtr (LPWORD lpIn)
{
  ULONG ul;

  ul = (ULONG) lpIn;
  ul +=3;
  ul >>=2;
  ul <<=2;
  return (LPWORD) ul;
}
```

Additional reference words: porting Win95 4.00 CreateDialogIndirectParam
KBCategory: kbprg kbprb
KBSubcategory: UsrDlg

PRB: CreateEllipticRgn() and Ellipse() Shapes Not Identical

PSS ID Number: Q83807

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When CreateEllipticRgn() is used to create a region in the shape of an ellipse and Ellipse() is called with the same parameters to draw an ellipse on the screen, the calculated region does not match the drawn ellipse identically.

CAUSE

Ellipse() includes the lower-right point of the bounding rectangle in its calculations, while the CreateEllipticRgn function excludes the lower-right point.

RESOLUTION

To draw a filled ellipse on the screen that matches an elliptic region, create the region with CreateEllipticRgn() and call FillRgn() to fill the region with the currently selected brush.

MORE INFORMATION

The region created by the CreateEllipticRgn() is slightly smaller than the elliptical area created by Ellipse(). Unfortunately, decreasing the width and height of the bounding rectangle by 1 pixel does not solve the problem. Although changing the parameters of the Ellipse() API in this way produces a smaller ellipse, the new ellipse does not match the region created with CreateEllipticRgn().

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic kbprb
KBSubcategory: GdiDrw

PRB: CreateFile Fails on Win32s w/ UNC Name of Local Machine

PSS ID Number: Q129543

The information in this article applies to:

- Microsoft Win32s version 1.2

SYMPTOMS
========

CreateFile() fails on Win32s if the file name is a UNC name that refers to the local machine.

CAUSE
=====

This is a limitation of Windows for Workgroups version 3.11. The same problem occurs with 16-bit Windows-based applications using OpenFile().

RESOLUTION
==========

Do not use a UNC name to open a file on the local machine.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

There is a similar limitation with Windows version 3.1 and LAN Manager. If you create a network drive and try to open a file on the same network share using a UNC name, it will fail. This also happens with 16-bit Windows-based applications.

Additional reference words: 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: CreateFile() Does Not Handle OEM Chars as Expected

PSS ID Number: Q129544

The information in this article applies to:

- Microsoft Win32s, versions 1.2 and 1.25a

SYMPTOMS

When CreateFile() is passed a filename string that contains an OEM character, the name of the file created is not as expected. In particular, a file created with CreateFile() cannot be opened with OpenFile() when OpenFile() is passed the same filename string.

For example, suppose that the ANSI filename string contains a lower-case e accent character (0x0e9). The file created by CreateFile() contains an upper-case E and the file created by OpenFile() contains an upper-case E accent character.

CAUSE

CreateFile() is implemented with calls to MS-DOS. MS-DOS converts the given file names to upper-case letters. OpenFile() is implemented with a thunk to the 16-bit Windows OpenFile(). OpenFile() converts the file name to upper-case before calling MS-DOS. The conversion that the 16-bit OpenFile() is doing is different from the conversion performed by MS-DOS for the OEM characters. The result is that different filenames are created for the same string passed to CreateFile() and OpenFile() if the name contains OEM characters.

RESOLUTION

To make the file name created by CreateFile() consistent with the file name created by OpenFile(), call AnsiUpper() on the file name string before calling CreateFile().

STATUS

This behavior is by design. This will not be changed in Win32s now, because it may break existing Win32-based applications.

MORE INFORMATION

The 16-bit OpenFile() will fail to find an existing file if the file contains OEM characters but not an explicit full path. This also occurs with the 32-bit OpenFile(), because it is thunked to the 16-bit OpenFile().

SearchFile() will also fail to find files if the file name contains OEM

characters or if the search path (lpszpath != NULL) contains OEM characters.

NOTE: In 16-bit Windows and Win32s 1.2 and earlier, OpenFile() returns the filename in OEM characters. However, the Win32 API documentation states that OpenFile() should return an ANSI string. Starting with the next version of Win32s, OpenFile() will return an ANSI string, as it does under Windows NT.

Additional reference words: 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: CreateProcess() of Windows-Based Application Fails

PSS ID Number: Q127860

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SYMPTOMS

When you spawn a 16-bit Windows-based application using CreateProcess() where neither lpApplicationName and lpCommandLine are NULL, WOW gives a popup saying:

> Cannot find file (or one of its components). Check to ensure the path and filename are correct and that all required libraries are available.

CAUSE

NTVDM expects the first token in the command line (lpCommandLine) to be the program name, although the Win32 subsystem does not. The current design will not be changed.

RESOLUTION

Make lpApplicationName NULL and put the full command line in lpCommandLine.

STATUS

This behavior is by design.

MORE INFORMATION

The documentation for CreateProcess() states:

> If the process to be created is an MS-DOS-based or Windows-based application, lpCommandLine should be a full command line in which the first element is the application name.

In this case (lpApplicationName is not NULL), lpCommandLine not only should be a full command line, but it must be a full command line.

Additional reference words: 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseProcThrd

PRB: Data Section Names Limited to Eight Characters

PSS ID Number: Q100292

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

SYMPTOMS

Data sections can be named by using #pragma data_seg. This method is commonly used so that the named data sections can be shared using the SECTIONS statement in the DEF file. However, if the length of the name specified in the pragma exceeds eight characters, then the section is not properly shared.

CAUSE

The linker does not support sections with longer names. The longer names require use of the COFF strings table, so the rewrite is not trivial.

MORE INFORMATION

Note that in addition, the first character of a section name must be a period. Therefore, the section name, as specified in both the pragma and the DEF file, can be a maximum of a period followed by seven characters.

For more information on the shared named-data section, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q89817
   TITLE   : How to Specify Shared and Nonshared Data in a DLL

Additional reference words: 3.10 3.50
KBCategory: kbtool kbprb
KBSubcategory: TlsMisc

PRB: DDEML Fails to Call TranslateMessage() in its Modal Loop

PSS ID Number: Q102576

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

During a synchronous transaction, DDEML causes the client to enter a modal loop until the transaction is processed. While DDEML dispatches messages appropriately, it fails to call TranslateMessage() while inside this modal loop. This problem does not apply to asynchronous transactions, where no such modal loop is entered.

SYMPTOMS

A common symptom of this problem is seen as the client processes user input while inside DDEML's modal loop in a synchronous transaction. WM_KEYDOWN and WM_KEYUP messages are received, with no corresponding WM_CHAR message for the typed character.

CAUSE

No WM_CHAR message is received because the WM_KEYDOWN message is never translated. For this to take place, a call to TranslateMessage() must be made inside the modal loop.

RESOLUTION

This limitation is by design. DDEML applications can work around this limitation by installing a WH_MSGFILTER hook, watching out for code == MSGF_DDEMGR.

The WH_MSGFILTER hook allows an application to filter messages while the system enters a modal loop, such as when a modal dialog box (code == MSGF_DIALOGBOX) or a menu (code == MSGF_MENU) is displayed; and similarly, when DDEML enters a modal loop in a synchronous transaction (code == MSGF_DDEMGR).

The Windows 3.1 Software Development Kit (SDK) DDEML\CLIENT sample demonstrates how to do this in DDEML.C's MyMsgFilterProc() function:

```
/*********************************************************************
```

```
 *
 *  FUNCTION: MyMsgFilterProc
 *
 *  PURPOSE:  This filter proc gets called for each message we handle.
 *            This allows our application to properly dispatch messages
 *            that we might not otherwise see because of DDEMLs modal
 *            loop that is used while processing synchronous transactions.
 *
****************************************************************************/

   DWORD FAR PASCAL MyMsgFilterProc( int nCode, WORD wParam,

                DWORD lParam)
   {
       wParam; // not used

    #define lpmsg ((LPMSG)lParam)

    if (nCode == MSGF_DDEMGR) {

            /* If a keyboard message is for the MDI, let the MDI client
             * take care of it. Otherwise, check to see if it's a normal
             * accelerator key. Otherwise, just handle the message as usual.
             */

            if ( !TranslateMDISysAccel (hwndMDIClient, lpmsg) &&
                 !TranslateAccelerator (hwndFrame, hAccel, lpmsg)){
                TranslateMessage (lpmsg);
                DispatchMessage (lpmsg);
            }
            return(1);
       }
       return(0);
   #undef lpmsg
   }
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrDde

PRB: DDEML with Excel Error: Remote Data Not Accessible

PSS ID Number: Q95982

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When data is linked between a Microsoft Excel spreadsheet and a DDEML server application, with both applications open, the following error message may appear:

```
Remote data not accessible; start application <SERVER FILENAME>.EXE?
```

CAUSE

Excel broadcasts an initiate to all windows, for every service/topic name pair it finds, in an attempt to initiate a conversation with the server application. (This can be easily verified by running DDESPY and watching Excel broadcast its initiates.) If Excel can't get a response, or gets a NACK (negative ACK), Excel attempts to EXEC() a new instance of the server application.

RESOLUTION

A DDEML server application should return TRUE, in response to the XTYP_CONNECT transaction it receives, for every service/topic name pair it supports. Refer to page 518 of the Microsoft Windows Software Development Kit (SDK) version 3.1 "Programmer's Reference, Volume 3: Messages, Structures, and Functions" for more information on the XTYP_CONNECT transaction.

MORE INFORMATION

The DDEML server application responds to Excel's initiate by sending the XTYP_CONNECT transaction to the DDE callback function of each server application, passing the service and topic names to the server.

If the server application fails to return TRUE to the service/topic name it supports, Excel concludes that it is trying to initiate a DDE link to an application that is not available, and brings up the message box above, thus giving the user an option to start the application.

Additional reference words: 3.50 3.51 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrDde

PRB: DDESpy Track Conversations Strings Window Empty

PSS ID Number: Q88197

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

In the DDESpy application, provided with version 3.1 of the Microsoft Windows Software Development Kit (SDK), DDE messages appear in the main window, while the Track Conversation Strings window remains empty.

CAUSE

Either the DDESpy application was started after the client and server applications, or neither application uses functions in the Dynamic Data Exchange Management Library (DDEML) to conduct DDE.

RESOLUTION

Start the DDESpy application before starting the client and server applications. Make sure that the client or the server (or both) uses the DDEML.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsSpy

PRB: Debugger Reports "WARNING: symbols checksum is wrong"

PSS ID Number: Q148220

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Microsoft Windows NT versions 3.51 and 4.00

SYMPTOMS

When debugging a Win32 application on the uniprocessor version of Windows NT, WinDbg displays warning messages that say:

```
*** WARNING: symbols checksum is wrong 0x0005e382 0x00062261 for
    D:\WINNT\symbols\dll\NTDLL.DBG.
```

The actual path for Ntdll.dbg will vary depending on where Windows NT is installed on your system. This warning may be generated for symbol files of other system files such as Kernel32.dll also.

CAUSE

WinDbg verifies that the checksum of the DLL matches the checksum stored in the corresponding debugging symbol (.dbg) file to warn you when your debugging symbols don't match the modules being debugged. If the symbols truly are mismatched, then WinDbg will show incorrect function names when you obtain stack traces and set breakpoints.

During installation, Windows NT runs with the multiprocessing kernel and system files. At the end of the installation procedure on uniprocessor Intel x86 machines, the setup program modifies the kernel and other system files to remove spin lock code needed only on multiprocessor systems.

Because the debugging symbol files were created with the multiprocessing system files, this modification causes the checksums of system files to differ from the checksums in the debugging symbol files. It does not change the file size of the system DLLs, any system code, or the addresses of system entry points. The symbol files are still correct, but because the checksums differ, WinDbg reports a warning.

RESOLUTION

The warning describes a benign condition, so you may safely ignore it.

STATUS

This behavior is by design.

Additional reference words: 4.00 3.51 KERNEL32

KBCategory: kbprg kbprb
KBSubcategory: TlsWindbg

PRB: Debuggers Cannot Reliably Use Debug Register Breakpoints

PSS ID Number: Q137199

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

Debug register breakpoints are breakpoints that trigger on write or read of a specific memory location. If you use debug register breakpoints in application debuggers such as CodeView, Visual C++, or Windbg, other applications in the system could fault.

CAUSE

Debug registers are not saved and restored on a per application basis in Windows 95; they are global to the system.

A debug register running at the same address (but in another memory context) could trigger that debug breakpoint. Because there may not be a debugger registered to handle the breakpoint in the second application, Windows 95 interprets this as an unhandled exception. In most cases, this causes Windows 95 to terminate the second application, even though it had nothing to do with the problem.

RESOLUTION

To work around this problem, use a system level debugger such as Soft-ICE or WDEB386, or avoid using debug register breakpoints.

STATUS

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbtool kbprb
KBSubcategory: TlsWindbg

PRB: Debugging an Application Driven by MS-TEST

PSS ID Number: Q100957

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

When an application driven by MS-TEST is being debugged by WinDbg or NTSD (for example, after an exception has occurred), both the application and the debugger hang.

CAUSE

The debugger is hooked and ends up hanging.

RESOLUTION

It is not possible to use NTSD or WinDbg to debug an application that is driven by MS-TEST. Use Dr. Watson (drwtsn32) instead. Note that you must turn off Visual Notification.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbprb
KBSubcategory: TlsWindbg

PRB: Debugging the Open Common Dialog Box in WinDbg

PSS ID Number: Q99952

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

When debugging an application that uses the Open common dialog box (created by the GetOpenFileName() function) in WinDbg, the program stops and the following information is displayed in the Command window:

```
   Thread Terminate: Process=0, Thread=2, Exit Code=1
```

CAUSE

The Open common dialog box causes a thread to be created. At this point in the debugging, that thread has terminated. By default, WinDbg halts whenever a thread terminates.

RESOLUTION

Execute the go command (type "g" at the command prompt). Execution will continue.

MORE INFORMATION

To prevent WinDbg from halting when a thread is terminated, select Debug from the Options menu and check "Go on thread terminate."

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbprb
KBSubcategory: TlsWindbg

PRB: Demand-started Service Causes Machine to Hang

PSS ID Number: Q152661

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK),for Windows NT, versions 3.50, 3.51

SYMPTOMS

A service is specified to start in the security context of a user. When the service is auto-started, it starts correctly. If it is demand-started (by the StartService API), the machine hangs. This state is demonstrated by the operating system becoming unresponsive. If you are starting the service from the service control panel applet, the hour-glass cursor fails to appear or, after a long delay, the "Attempting to Start Service" dialog box appears.

CAUSE

The problem is caused by an incorrectly written third-party network provider. The service control manager must check with each installed network provider in order to authenticate that the user name and password for the service are valid. If a network provider has trouble validating the user's credentials, the service control manager fails to start the service.

RESOLUTION

Remove the third-party network provider from the system. Please refer to your network provider documentation for more information on how to remove it from the system. Contact the network provider vendor for an update that resolves the problem.

STATUS

This behavior is by design.

Additional reference words: 3.50 3.51
KBCategory: kbprg kb3rdparty kbprb
KBSubcategory: bseService

PRB: Description for Event ID Could Not Be Found

PSS ID Number: Q129003

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5
- Microsoft Visual C++, 32-bit Edition, version 2.0

SYMPTOMS

After building and executing the Win32 LOGGING sample that accompanies Visual C++ version 2.0 and the Microsoft Developer's Network Library CD, you find that the descriptions for the logged events contain the following:

> The Description for Event ID (#) in Source (application name) could not be found. It contains the following string(s):

CAUSE

The resource file containing the descritions that correspond to the Event IDs are not bound to the MESSAGES.DLL.

RESOLUTION

Add the MESSAGES.RC file to the Visual C++ version 2.0 project:

1. Run MAKEMC.BAT first. MAKEMC.BAT creates the MSG00001.BIN, MESSAGES.RC, and MESSAGES.H files.

2. Open Visual C++ version 2.0 and add MESSAGES.RC to the MESSAGES project (MESSAGES.MAK).

3. Rebuild both the MESSAGES and the LOG projects.

NOTE: If you ran the sample a couple of times from different places, you may need to delete the LOG registry key if it contains an incorrect location for the MESSAGES.DLL file.

Additional reference words: 3.50 3.51 Event Logging Log MSDN
KBCategory: kbprg kbprb
KBSubcategory: BseMisc

PRB: Device and TrueType Fonts Rotate Inconsistently

PSS ID Number: Q82932

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

In an application for the Microsoft Windows graphical environment that uses a mapping mode other than MM_TEXT and a text alignment other than the default (TA_LEFT, TA_TOP), device fonts and TrueType fonts rotate in the opposite directions. Device fonts may exhibit other unusual behaviors. Differences in rotated text may appear on the screen or in printed output.

RESOLUTION

To create a font that rotates based on the coordinate system in which the font is used, the application must set the CLIP_LH_ANGLES (0x10) bit in the lfClipPrecision field of the LOGFONT data structure. This technique is backward-compatible with Windows 3.0 because the CLIP_LH_ANGLES bit is ignored in version 3.0.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic kbprb
KBSubcategory: GdiTt

PRB: DeviceIoControl Int 13h Does Not Support Hard Disks

PSS ID Number: Q137176

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SYMPTOMS

Win32-based applications running under Windows 95 use CreateFile to open VWIN32 and then use DeviceIoControl with the VWIN32_DIOC_DOS_INT13 flag to perform low-level BIOS disk functions. The functions work on floppy disks but always fail on hard disks.

CAUSE

Windows 95, like previous versions of Windows, does not support calling BIOS disk functions to gain access to hard disks from protected mode Windows- and Win32-based applications. The cause of this lack of support for hard disks is that BIOSXLAT.VXD does not translate BIOS requests from protected-mode into V86 mode, and this causes the ROM BIOS to be called with an invalid address.

RESOLUTION

To work around this problem, Win32-based applications must thunk to a Windows 16-bit DLL and have that DLL call the DOS Protected Mode Interface (DPMI) Simulate Real Mode Interrupt function to call Int 13h BIOS disk functions on hard disks. When you use DPMI to call Int 13h BIOS disk functions, you bypass BIOSXLAT.VXD and call the real-mode BIOS. Note that you cannot call DPMI from 32-bit code.

STATUS

This behavior is by design.

MORE INFORMATION

The following example code shows how a Win32-based application running under Windows 95 can read the first physical sector of a hard disk, which contains the disk's partition tables. This code requires a flat thunk, so parts of it must be placed in a Win32 DLL and other parts must be placed in a Windows 16-bit DLL. The relevant code for each DLL is labeled.

Execution starts from a Win32-based application calling the Win32 DLL and proceeds as follows. The Win32 DLL function, CallReadPhysicalSector1, calls

the 16-bit DLL's ReadPhysicalSector1 function by way of a thunk. Then ReadPhysicalSector1, which is exported by the 16-bit DLL, calls DPMI to call the BIOS Read Track function to read the first sector of the specified hard disk. After ReadPhysicalSector1 returns, the contents of the first sector of the hard disk, including the hard disk's partition table and bootstrap program, will be in the buffer pointed to by lpBuff.

Code Sample

```
//--------------------------------------------------------------------
// Code in the 32-bit DLL

// Prototype for function in 16-bit DLL.
BOOL FAR PASCAL ReadPhysicalSector1 (BYTE   bDrive,
                                    LPBYTE lpBuffer,
                                    DWORD  cbBuffSize);

/*--------------------------------------------------------------------
 CallReadPhysicalSector1()

 Reads the first sector of the first hard disk.

 Return Value
    Returns TRUE if the first sector was read, and FALSE if it
    wasn't.
--------------------------------------------------------------------*/
__declspec(dllexport) BOOL WINAPI CallReadPhysicalSector1()
{
    char lpBuff[512];
    BOOL fResult;

    // Read the first sector of the first hard disk.
    fResult = ReadPhysicalSector1 (0x80, lpBuff, 512);

    if (fResult)
       {
        // lpBuff contains the sector data.  Use it here
       }

    return fResult;
}


//--------------------------------------------------------------------
// Thunk Script

enablemapdirect3216 = true;

typedef unsigned long   DWORD;
typedef unsigned char   BYTE, * LPBYTE;
typedef bool            BOOL;

BOOL ReadPhysicalSector1 (BYTE   bDrive,
```

```
                          LPBYTE lpBuffer,
                          DWORD  cbBuffSize)
{
   lpBuffer = inout;
}


//--------------------------------------------------------------------
// Code in the 16-bit DLL

// Converts two BYTEs into a WORD.  This is useful for working with
// a RMCS, but was not provided in WINDOWS.H.

//#define MAKEWORD(low, high) \
           ((WORD)((((WORD)(high)) << 8) | ((BYTE)(low))))

#define SECTOR_SIZE 512       // Size, in bytes, of a disk sector
#define CARRY_FLAG  0x0001

typedef BYTE FAR *LPBYTE;

typedef struct tagRMCS
{
   DWORD edi, esi, ebp, RESERVED, ebx, edx, ecx, eax;
   WORD  wFlags, es, ds, fs, gs, ip, cs, sp, ss;
} RMCS, FAR* LPRMCS;

BOOL FAR PASCAL SimulateRM_Int (BYTE bIntNum, LPRMCS lpCallStruct);
void FAR PASCAL BuildRMCS (LPRMCS lpCallStruct);
BOOL FAR PASCAL __export ReadPhysicalSector1 (BYTE   bDrive,
                                              LPBYTE lpBuffer,
                                              DWORD  cbBuffSize);

/*--------------------------------------------------------------------
  ReadPhysicalSector1()

  Calls DPMI to call the BIOS Int 13h Read Track function to read the
  first physical sector of a physical drive. This function is used to
  read partition tables, for example.

  Parameters
     bDrive
        The Int 13h device unit,
           0x00 for floppy drive 0
           0x00 for floppy drive 1
           0x80 for physical hard disk 0
           0x81 for physical hard disk 1
           etc.

     lpBuffer
        Pointer to a buffer that receives the sector data.  The buffer
        must be at least SECTOR_SIZE bytes long.

     cbBuffSize
        Actual size of lpBuffer.
```

```
  Return Value
     Returns TRUE if the first sector was read into the buffer pointed
     to by lpBuffer, or FALSE otherwise.

  Assumptions
     Assumes that sectors are at least SECTOR_SIZE bytes long.
-------------------------------------------------------------------*/

BOOL FAR PASCAL __export ReadPhysicalSector1 (BYTE   bDrive,
                                              LPBYTE lpBuffer,
                                              DWORD  cbBuffSize)
{
   BOOL   fResult;
   RMCS   callStruct;
   DWORD  gdaBuffer;     // Return value of GlobalDosAlloc().
   LPBYTE RMlpBuffer;    // Real-mode buffer pointer
   LPBYTE PMlpBuffer;    // Protected-mode buffer pointer

   /*
     Validate params:
        bDrive should be int 13h device unit -- let the BIOS validate
           this parameter -- user could have a special controller with
           its own BIOS.
        lpBuffer must not be NULL
        cbBuffSize must be large enough to hold a single sector
   */

   if (lpBuffer == NULL || cbBuffSize < SECTOR_SIZE)
      return FALSE;

   /*
     Allocate the buffer that the Int 13h function will put the sector
     data into. As this function uses DPMI to call the real-mode BIOS, it
     must allocate the buffer below 1 MB, and must use a real-mode
     paragraph-segment address.

     After the memory has been allocated, create real-mode and
     protected-mode pointers to the buffer. The real-mode pointer
     will be used by the BIOS, and the protected-mode pointer will be
     used by this function because it resides in a Windows 16-bit DLL,
     which runs in protected mode.
   */

   gdaBuffer = GlobalDosAlloc (cbBuffSize);

   if (!gdaBuffer)
      return FALSE;

   RMlpBuffer = (LPBYTE)MAKELONG(0, HIWORD(gdaBuffer));
   PMlpBuffer = (LPBYTE)MAKELONG(0, LOWORD(gdaBuffer));

   /*
     Initialize the real-mode call structure and set all values needed
     to read the first sector of the specified physical drive.
```

```
   */

   BuildRMCS (&callStruct);

   callStruct.eax = 0x0201;               // BIOS read, 1 sector
   callStruct.ecx = 0x0001;               // Sector 1, Cylinder 0
   callStruct.edx = MAKEWORD(bDrive, 0);  // Head 0, Drive #
   callStruct.ebx = LOWORD(RMlpBuffer);   // Offset of sector buffer
   callStruct.es  = HIWORD(RMlpBuffer);   // Segment of sector buffer

   /*
      Call Int 13h BIOS Read Track and check both the DPMI call
      itself and the BIOS Read Track function result for success.  If
      successful, copy the sector data retrieved by the BIOS into the
      caller's buffer.
   */

   if (fResult = SimulateRM_Int (0x13, &callStruct))
      if (!(callStruct.wFlags & CARRY_FLAG))
         {
         _fmemcpy (lpBuffer, PMlpBuffer, (size_t)cbBuffSize);
         fResult = TRUE;
         }
      else
         fResult = FALSE;

   // Free the sector data buffer this function allocated
   GlobalDosFree (LOWORD(gdaBuffer));

   return fResult;
}


/*---------------------------------------------------------------------
  SimulateRM_Int()

  Allows protected mode software to execute real mode interrupts such
  as calls to DOS TSRs, DOS device drivers, etc.

  This function implements the "Simulate Real Mode Interrupt" function
  of the DPMI specification v0.9.

  Parameters
     bIntNum
        Number of the interrupt to simulate

     lpCallStruct
        Call structure that contains params (register values) for
        bIntNum.

  Return Value
     SimulateRM_Int returns TRUE if it succeeded or FALSE if it
     failed.

  Comments
```

```
     lpCallStruct is a protected-mode selector:offset address, not a
     real-mode segment:offset address.
-------------------------------------------------------------------*/

BOOL FAR PASCAL SimulateRM_Int (BYTE bIntNum, LPRMCS lpCallStruct)
   {
   BOOL fRetVal = FALSE;      // Assume failure

   _asm {
         push di

         mov  ax, 0300h         ; DPMI Simulate Real Mode Int
         mov  bl, bIntNum       ; Number of the interrupt to simulate
         mov  bh, 01h           ; Bit 0 = 1; all other bits must be 0
         xor  cx, cx            ; No words to copy
         les  di, lpCallStruct
         int  31h                  ; Call DPMI
         jc   END1                 ; CF set if error occurred
         mov  fRetVal, TRUE
      END1:
         pop di
        }
   return (fRetVal);
   }

/*-------------------------------------------------------------------
  BuildRMCS()

  Initializes a real mode call structure to contain zeros in all its
  members.

  Parameters:
     lpCallStruct
        Points to a real mode call structure

  Comments:
     lpCallStruct is a protected-mode selector:offset address, not a
     real-mode segment:offset address.
-------------------------------------------------------------------*/

void FAR PASCAL BuildRMCS (LPRMCS lpCallStruct)
   {
   _fmemset (lpCallStruct, 0, sizeof (RMCS));
   }

REFERENCES
==========

Information on using DPMI may be found in the DOS Protected Mode
Interface (DPMI) Specification Version 0.9.

Flat thunks are documented in "The Microsoft Programmer's Guide To
Windows 95" topic in the Win32 Software Development Kit online
documentation.
```

Additional reference words: 4.00 Windows 95 sector table physical
KBCategory: kbprg kbprb
KBSubcategory: BseFileio

PRB: Dialog Box and Parent Window Disabled

PSS ID Number: Q11337

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When an application uses one of the DialogBox family of functions to create a modal dialog box, both the parent window and the dialog box are disabled (unable to accept keyboard or mouse input).

CAUSE

In the application's resource file, the dialog box resource has the WS_CHILD style.

RESOLUTION

To avoid this problem, use the WS_POPUP style instead of the WS_CHILD style.

MORE INFORMATION

When an application creates a modal dialog box using one of the DialogBox family of functions, Windows disables the dialog box's parent window. If the parent window has any child windows, the child windows are also disabled.

An application can use the WS_CHILD style for dialog boxes created by one of the CreateDialog family of functions. However, problems and inconsistencies arise if the application uses the IsDialogMessage function to process dialog box input for either the parent or the child.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
CreateDialogIndirect CreateDialogIndirectParam CreateDialogParam
DialogBoxIndirect DialogBoxIndirectParam DialogBoxParam
KBCategory: kbui kbprb
KBSubcategory: UsrDlgs

PRB: Dialog Editor Does Not Modify RC File Dialog Box Resource

PSS ID Number: Q32019

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.1 and 3.5

SYMPTOMS

If the Windows Dialog Editor (DIALOG.EXE) is used to edit a dialog box, when the associated application is built, the previous version of the dialog box is used.

CAUSE

The dialog resource used by the program is stored in the application's resource file (which has the extension RC). The Dialog Editor stores the updated dialog box resource in a file with the .DLG extension.

RESOLUTION

Modify the resource file to use the RCINCLUDE statement to include the updated dialog box resource, as follows:

```
RCINCLUDE DIALOG.DLG
```

MORE INFORMATION

Suppose an application is written that uses a dialog box. The developer creates a dialog box template manually in the application's resource file. After building the application, the developer decides to modify the dialog box using the Dialog Editor.

When the Dialog Editor edits an existing dialog box, it reads the application's compiled resources stored in a compiled resource file (with the extension .RES). However, when the developer saves any changes, the Dialog Editor creates a file with the .DLG extension.

When the application is built for a second time, the Resource Compiler uses the original definition for the dialog box stored in the resource file. The resolution provided above avoids this problem because the Resource Compiler always includes the latest version of the dialog box template.

For more information about this topic, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q40958
TITLE : PRB: DIALOG.EXE Reads Compiled .RES Files, Not .DLG Files

Additional reference words: 3.00 3.10 3.50
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsDlg

PRB: Dialog Editor Does Not Retain Unsupported Styles

PSS ID Number: Q74264

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

The Windows Dialog Editor does not retain control/dialog styles that it does not support.

RESOLUTION

The styles in question must be manually added to the dialog script before using the resource compiler to compile the resource script.

MORE INFORMATION

Any style that cannot be explicitly set in the Dialog Editor Styles dialog box will not be present in the script file generated by the the Dialog Editor. For example, the edit control style, ES_OEMCONVERT is not available in the Dialog Editor Styles dialog box. If this style is manually added to the dialog script, compiled, and then loaded into the Dialog Editor, the ES_OEMCONVERT style will be removed.

Any styles that are not supported by the Dialog Editor must be manually added to the dialog script. To modify the dialog script, use a text editor to combine the desired styles using the bitwise OR operator("|"). This modified script should then be included into the application's resource script. If the application's .RES file is later loaded into the Dialog Editor, the unsupported styles will be removed and will need to be added again as described above.

The following is a list of styles that are known not to be supported by the Dialog Editor:

Style

CBS_OEMCONVERT
ES_OEMCONVERT
SBS_BOTTOMALIGN
SBS_LEFTALIGN
SBS_RIGHTALIGN
SBS_SIZEBOX
SBS_SIZEBOXBOTTOMRIGHTALIGN
SBS_SIZEBOXTOPLEFTALIGN

SBS_TOPALIGN
SS_LEFTNOWORDWRAP
SS_NOPREFIX
SS_USERITEM

In addition to the above control styles, any menu definitions added to the dialog script via the MENU resource statement will be deleted by the Dialog Editor.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool kbprb
KBSubcategory: TlsDlg

PRB: DIALOG.EXE Reads Compiled .RES Files, Not .DLG Files

PSS ID Number: Q40958

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.1 and 3.5

SYMPTOMS

Create a dialog box using DIALOG.EXE, the Dialog Editor. From MS-DOS, change an attribute (for example, position of control) of the dialog box by modifying the .DLG file produced by DIALOG.EXE. Invoke DIALOG.EXE again, the changes that were made are not evident.

CAUSE

The Dialog Editor produces .DLG and .RES files for the dialog box created. When using the Dialog Editor to modify an existing dialog box, the Dialog Editor will look at the .RES file for information about the makeup of the dialog box. The .DLG file is completely ignored at this point. This is why the Dialog Editor does not recognize any .DLG changes.

RESOLUTION

For the modifications to be recognized, the Resource Compiler (RC) must be used with the -r switch to compile the .DLG file. The new .RES file can then be loaded into the Dialog Editor and the changes will be recognized.

Additional reference words: 3.00 3.10 3.50
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsDlg

PRB: DialogBox() Call w/ Desktop Window Handle Disables Mouse

PSS ID Number: Q129597

The information in this article applies to:

- Microsoft Win32s, version 1.2

SYMPTOMS
========

Under Windows NT and Windows 95, you can call DialogBox() using the return from GetDesktopWindow() as the parent window handle. However, doing the same thing under Win32s will disable the mouse. Clicking any mouse button results in a beep, and you must reboot the computer to enable the mouse again.

CAUSE
=====

This is a bug in Windows, not Win32s. The same thing occurs from a 16-bit Windows-based application. Internally, Windows disables the parent window by calling EnableWindow(hwndParent, FALSE). If the handle is the desktop window handle, the desktop window is disabled. This disables the mouse as well, due to the bug in the Windows code.

RESOLUTION
==========

Do not call DialogBox() with the desktop window handle as the parent window handle.

Additional reference words: 1.20 HWND_DESKTOP
KBCategory: kprg kbprb
KBSubcategory: W32s

PRB: Display Problems with Win32s and the S3 Driver

PSS ID Number: Q117153

The information in this article applies to:

- Microsoft Win32s versions 1.1, 1.15, and 1.2

SYMPTOMS

Applications running on Windows 3.1 exhibit no problems when using an S3 driver, with a resolution of 1024x768 and 256 colors. However, the display is corrupted when Win32-based applications, such as FreeCell, run on Win32s in the same configuration.

CAUSE

This is a known problem with the S3 driver.

A general protection (GP) fault occurs when the display driver is performing a bit-block transfer (BitBlt). This happens whether or not Win32s is running. However, when Win32s is not running, Windows recovers from the faults. When a Win32-based application is running, Win32s catches all exceptions and transfers control to the nearest try/except frame. As a result, the BitBlt is interrupted.

RESOLUTION

Certain S3 drivers which exhibit these problems can be made to work with Win32s by making the following edit to your SYSTEM.INI file before running any Win32-based applications:

In the SYSTEM.INI file, you will find an entry in the [display] section

```
aperture-base=100
```

Change this entry to

```
aperture-base=0
```

Restart Windows and the display problems will no longer occur.

If this does not help, obtain the latest S3 drivers. It is reported that S3 driver version 1.3 does not have this problem.

Additional reference words: 1.10 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: Dithered Brushes Are Not Supported in DIB Sections

PSS ID Number: Q137370

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.5, 3.51, 4.0

SYMPTOMS
========

Dithered brushes do not work in device contexts that have DIB sections selected into them. If you create an HBITMAP using CreateDIBSection(), select it into a memory DC, and then attempt to use a dithered brush returned by CreateSolidBrush() to draw into the DIB section, the brush will not be drawn with a dithered color.

CAUSE
=====

To handle dithered brushes, the GDI relies on certain colors being available in particular positions in the palette. DIB sections do not have to have these colors in their color table, so the GDI cannot rely on the necessary colors being available for dithering.

STATUS
======

This behavior is by design.

Additional reference words: 4.00 3.50 bitmap solid fillrect hbrush
KBCategory: kbgraphic kbprb
KBSubcategory: W32

PRB: DLL Load Fails Under Win32s When Apps Share a DLL

PSS ID Number: Q131224

The information in this article applies to:

- Microsoft Win32s versions 1.2 and 1.25a

SYMPTOMS

LoadLibrary fails under Win32s in the following situation:

1. App1 is executed and loads MYDLL.DLL.

2. App2 is executed and loads MYDLL.DLL as well.

3. App1 is terminated and unloads MYDLL.DLL.

App2 can GP fault when MYDLL.DLL is accessed. In addition, if App1 is restarted and attempts to load MYDLL.DLL, the call to LoadLibrary fails and GetLastError reports ERROR_DLL_INIT_FAILED.

This problem does not occur under Windows NT or Windows 95.

CAUSE

By default, Visual C++ create a DLL that statically links to the C Run-time (CRT). This version of the CRT is not compatible with Win32s. Problems occur when global data for the CRT is initialized, because of the shared address space under Windows.

The static CRT library uses global variables to manage memory allocations. In the scenario above, App2 can fault when allocating memory, if the global variables used to access memory refer to memory that was allocated App1 which has since terminated, because the allocated memory was returned to the heap.

RESOLUTION

Use the /MD (Multithreaded using CRT in a DLL) option when compiling the DLL and add the MSVCRT.LIB import library to the library list. Include the Win32s version of MSVCRT20.DLL which is included in Visual C++ 2.x (it is redistributable) in your project. Then the DLL will use the DLL version of the CRT libraries that is compatible with Win32s and the problem will not occur.

MORE INFORMATION

When a DLL that uses the CRT is loaded into memory, all global variables

for the DLL and for the CRT libraries are initialized. Under Windows NT and Windows 95, the application is given its own copy of the global data for the DLL. When other applications use the same DLL, they each receive their own copy of the global variables as well. This eliminates conflicts, because the data is not shared.

Under Win32s, DLLs are loaded into the same shared memory space and all global variables for a DLL are shared. This means that the CRT global data is also shared. The version of MSVCRT20.DLL that targets Win32s was written to take this into account and avoid conflicts.

The Windows NT/Windows 95 version of MSVCRT20.DLL can be found in the MSVC20\REDIST directory of the CD. The Win32s version can be found in the WIN32S\REDIST directory.

Additional reference words: 1.20 2.00
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: Dotted Line Displays as Solid Line

PSS ID Number: Q24179

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When an application creates a dotted line using a dotted pen and the R2_NOT mode, a solid line is drawn on the screen.

CAUSE

The background mode is OPAQUE. This is the default background mode. In this mode, the line is painted with the background color first, followed by the pen.

RESOLUTION

Use the SetBkMode() function to set the mode to TRANSPARENT. In this mode, only the pen is used; the background is not disturbed.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic kbprb
KBSubcategory: GdiPnbr

PRB: Edit Control Margins too Large in Windows 95

PSS ID Number: Q138419

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS
========

After you change the font in an edit control in Windows 95, the left and right margins are unusually large.

CAUSE
=====

When the edit control receives the WM_SETFONT message, it calculates the left and right margins based on the average character width of the new font. The control does not do this when it is created. This results in the margins changing drastically when the font is changed.

RESOLUTION
==========

Get the margins of the edit control before sending the WM_SETFONT message, and then set the margins back to their original sizes. The following code fragment demonstrates how to do this:

```
{
DWORD dwMargins;

dwMargins = SendDlgItemMessage(  hWnd,
                                 IDC_EDIT,
                                 EM_GETMARGINS,
                                 0,
                                 0);

SendDlgItemMessage(  hWnd,
                     IDC_EDIT,
                     WM_SETFONT,
                     (WPARAM)<new font handle>,
                     0);

SendDlgItemMessage(  hWnd,
                     IDC_EDIT,
                     EM_SETMARGINS,
                     EC_LEFTMARGIN | EC_RIGHTMARGIN,
                     MAKELPARAM(LOWORD(dwMargins), HIWORD(dwMargins)));

}
```

STATUS
======

This behavior is by design.

Additional reference words: 1.30 4.00 Windows 95
KBCategory: kbprg kbui kbprb kbcode
KBSubcategory: UsrCtl W32s

PRB: Editing Labels in a TreeView Gives WM_COMMAND|IDOK Errors

PSS ID Number: Q130692

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS

WM_COMMAND|IDOK errors are received while editing labels in a TreeView control.

CAUSE

While editing labels in a TreeView control, the edit control created by the TreeView control can, and usually does, have an identifier of 1. This identifier is the same as IDOK. This can cause the parent window or dialog box to receive WM_COMMAND messages with an identifier of 1. Then the TreeView control passes on the EN_UPDATE and EN_CHANGE notifications from the edit control to the TreeView's parent.

This was a design decision made to meet system requirements and cannot be changed. If the parent window is going to perform some action in response to a command with an identifier of 1, this problem can occur. This problem is especially significant in dialog boxes that use the standard IDOK for a command button control.

RESOLUTION

Avoid using command and control identifiers with an identifier of 1 (IDOK). To be safe, the application should not use any identifiers less than 100 when used in conjunction with a TreeView control.

Another way to avoid this problem is to check the notification codes in the WM_COMMAND messages. Then respond only to the proper notification codes such as BN_CLICKED.

STATUS

This behavior is by design.

Additional reference words: 1.30 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrCtl W32s

PRB: EndPage() Returns -1 When Banding

PSS ID Number: Q118873

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When an application that implements banding calls EndPage(), EndPage() returns SP_ERROR (-1).

CAUSE

EndPage() returns -1 if there has been no corresponding call to StartPage(). Windows keeps track of whether StartPage() has been called by maintaining an internal flag that is set when StartPage() is called and then is cleared when EndPage() is called.

This flag is also cleared when the NEXTBAND escape is called and there are no more bands on the page to be printed. At this point, Windows clears the internal flag and tells the device that a page has been finished. Because the internal flag has been cleared, a subsequent call to EndPage() returns -1.

RESOLUTION

Though EndPage() returns -1 when it is called from printing code that implements banding, it does no harm. An application can safely call StartPage() and EndPage() when banding and ignore the -1 error returned from EndPage().

NOTE: It is not recommended that a Win32-based application use banding. Windows NT, spools in a journal file and Windows 95 spools in an enhanced metafile, so all GDI calls are supported without banding.

Additional reference words: 3.10 3.50 4.00 NEWFRAME
KBCategory: kbprint kbprb
KBSubcategory: GdiPrn

PRB: Error 1 (NRC_BUFLEN) During NetBIOS Send Call

PSS ID Number: Q124879

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

When making a NetBIOS Send call, the call with fail with error code 1 (NRC_BUFLEN). This error code indicates that the buffer length is invalid.

CAUSE

NRC_BUFLEN will be returned if the buffer length specified in the NetBIOS Control Block (NCB) is incorrect. Less obvious is the fact that this error will also be returned if the buffer pointed to by the NCB is protected from write operations.

RESOLUTION

Although the NetBIOS code does not write to the buffer supplied in the NCB, write access is required. You can solve the problem by changing the protection on your buffer to include write access.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork kbnetwork kberrmsg
KBSubcategory: NtwkNetbios

PRB: Error Message Box Returned When DLL Load Fails

PSS ID Number: Q117330

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

Under Windows NT, when you load a DLL, a message box titled "Invalid DLL Entrypoint" is displayed and has the following text:

> The dynamic link library <name> is not written correctly.
> The stack pointer has been left in an inconsistent state.
> The entry point should be declared as WINAPI or STDCALL.
> Select YES to fail the DLL load. Select NO to continue
> execution. Selecting NO may cause the application to operate
> incorrectly.

Under Windows 95, the message box is titled "Error starting program" and the text is:

> The <dll file name> file cannot start. Check the file to
> determine the problem.

The user is not given a choice to continue, only an OK button. Pressing the OK button fails program load.

CAUSE
=====

The system expects DLL entrypoints to use the _stdcall convention. If you use the _cdecl convention, the stack is not properly restored and subsequent calls into the DLL can cause a general protection fault (GPF).

This error message is new to Windows NT, version 3.5. Under Windows NT, version 3.1, the DLL is loaded without an error message, but the application usually causes a GPF when calling a DLL routine.

RESOLUTION
==========

Correct the prototype of your entrypoint. For example, if your entrypoint is as follows:

```
BOOL DllMain( HANDLE hDLL, DWORD dwReason, LPVOID lpReserved)
```

change it to the following:

```
   BOOL WINAPI DllMain( HANDLE hDLL, DWORD dwReason, LPVOID lpReserved)
```

Then, link with the following linker option to specify the entry point if you are using the C run-time:

```
   -entry:_DllMainCRTStartup$(DLLENTRY)
```

MORE INFORMATION
==========

If you are using the Microsoft C run-time, you need to use the entry point given in the RESOLUTION section in order to properly initialize the C run-time. For additional information, please see the following article in the Microsoft Knowledge Base:

```
   ARTICLE-ID: Q94248
   TITLE     : Using the C Run Time
```

REFERENCES
==========

For more information on the DLL entrypoint, please search on the topic "DllEntryPoint" (without the quotes) in the Win32 API help file.

Additional reference words: 3.50 4.00 libmain
KBCategory: kbprg kbprb
KBSubcategory: BseDll

PRB: Error on Win32s: R6016 - not enough space for thread data

PSS ID Number: Q126709

The information in this article applies to:

- Microsoft Win32s, versions 1.2 and 1.25

SYMPTOMS

Spawning and closing an application repeatedly succeeds around 60 times, then the spawn fails with this error:

R6016 - not enough space for thread data

CAUSE

The thread local storage (TLS) is not freed by the system.

The failure occurs only if there is another Win32-based application active while you are doing the spawns. The message itself is not generated by Win32s. It is generated by the Microsoft C Run-time (CRT) libraries LIBC.LIB and LIBCMT.LIB.

RESOLUTION

In Win32s version 1.25, TLS indices are freed during module cleanup. The TLS index is owned by the application's main module, so that it is freed when the application terminates. This solves the problem for LIBC and LIBCMT.

There is a similar problem with MSVCRT20.DLL. This DLL version of the CRT allocates a new TLS index each time a process attaches to it. MSVCRT20 doesn't free the TLS indices when unloading. The system should free them. If only one application uses MSVCRT20 at a time, then the application can be spawned successfully up to about 60 times on Win32s version 1.20. On Win32s version 1.25, there is no limitation.

If there is already an active application that uses MSVCRT20, it is not possible to spawn and close a second application that uses MSVCRT20 more than about 60 times under Win32s version 1.25. This is because MSVCRT20 allocates a TLS index each time a process attaches to it. Win32s will free all of the TLS indices only when MSVCRT20 is unloaded.

MORE INFORMATION

On Win32s, TLS allocation should be done once and not per process. Each process can use the index to store per-process data, just as a thread uses a TLS index on Windows NT. This is easy to do, because DLL data is shared between all processes under Win32s.

The following example demonstrates how to do the TLS allocation once on Win32s:

```
   BOOL APIENTRY DllMain(HINSTANCE hinstDll, DWORD fdwReason,
           LPVOID lpvReserved)
   {
       static BOOL fFirstProcess = TRUE;
       BOOL fWin32s = FALSE;
       DWORD dwVersion = GetVersion();
       static DWORD dwIndex;

       if ( !(dwVersion & 0x80000000) && LOBYTE(LOWORD(dwVersion))<4 )
           fWin32s = TRUE;

       if (dwReason == DLL_PROCESS_ATTACH) {
           if (fFirstProcess || !fWin32s) {
               dwIndex = TlsAlloc();
            }
            fFirstProcess = FALSE;
       }
       .
       .
       .
   }
```

Additional reference words: 1.20
KBCategory: kbprg kbcode kbprb
KBSubcategory: W32s

PRB: Error with GetOpenFileName() and OFN_ALLOWMULTISELECT

PSS ID Number: Q99338

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.5

SYMPTOMS
========

Assume that the GetOpenFileName() function is called with the Flags parameter set to OFN_ALLOWMULTISELECT (allow for multiple selections in the list box) and OFN_NOVALIDATE (allow for invalid characters in the filename), the lpstrFile parameter points to a 2K buffer, and the corresponding nMaxFile is set appropriately to 2048.

When the GetOpenFileName() function call returns, the complete selection in the list box is not copied to the lpstrFile buffer, but the string in the buffer is truncated.

CAUSE
=====

This is a problem with the GetOpenFileName() function in the current version of COMMDLG.DLL in that the OFN_NOVALIDATE flag cannot be used when multiple selections are allowed.

RESOLUTION
==========

The following are two suggested solutions to this problem:

One solution for this problem is to not use the OFN_NOVALIDATE flag with the OFN_ALLOWMULTISELECT flag. That is, if only the OFN_ALLOWMULTISELECT flag is used, then multiple selections will be copied properly to the text buffer. Note that there is a buffer size limit of 2K bytes for the lpstrFile buffer, and characters are truncated after 2K bytes when the OFN_ALLOWMULTISELECT flag is used.

There is another solution, if both the OFN_ALLOWMULTISELECT and OFN_NOVALIDATE flags must be used simultaneously with GetOpenFileName(). Note that the entire string is always copied into the edit control even though the text gets truncated during the process of copying the text to the lpstrFile buffer. Therefore, one could write a hook procedure and read the entire string from the edit control and use it appropriately instead of using the lpstrFile buffer.

Additional reference words: 3.10 3.50

KBCategory: kbui kbprb
KBSubcategory: UsrCmnDlg

PRB: ERROR_INVALID_PARAMETER from WriteFile() or ReadFile()

PSS ID Number: Q110148

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS

The WriteFile() or ReadFile() function call may fail with the error ERROR_INVALID_PARAMETER if you are operating on a named pipe and using overlapped I/O.

CAUSE

A possible cause for the failure is that the Offset and OffsetHigh members of the OVERLAPPED structure are not set to zero.

RESOLUTION

Set the Offset and OffsetHigh members of your OVERLAPPED structure to zero.

STATUS

This behavior is by design. The online help for both WriteFile() and ReadFile() state that the Offset and OffsetHigh members of the OVERLAPPED structure must be set to zero or the functions will fail.

MORE INFORMATION

In many cases the function calls may succeed if you do not explicitly set OVERLAPPED.Offset and OVERLAPPED.OffsetHigh to zero. However, this is usually either because the OVERLAPPED structure is static or global and therefore is initialized to zero, or the OVERLAPPED structure is automatic (local) and the contents of that location on the stack are already zero. You should explicitly set the OVERLAPPED.Offset and OVERLAPPED.OffsetHigh structure members to zero.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg kbprb
KBSubcategory: BseFileio

PRB: Errors When Windbg Switches Not Set for Visual C++ App

PSS ID Number: Q131111

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SYMPTOMS

If two switches are not set correctly, Windbg gives the following error message when the module is loaded:

```
(symbol format not supported)
```

In addition, Windbg gives the following error message for any breakpoints set in the module:

```
Unresolved Breakpoint
```

CAUSE

For Windbg to understand the symbolic format generated from Microsoft Visual C++ version 2.0, two linker switches have to be set correctly:

- /DEBUG is set in the Project Settings dialog box, under the Link tab. Choose the Debug category. Then select the Generate Debug Info check box, and choose Microsoft Format.

- /PDB:none is set in the Project Settings dialog box, under the Link tab. Choose the Customize category. Then clear the Use Program Database check box.

STATUS

This behavior is by design.

Additional reference words: 3.50
KBCategory: kbtool kbprb
KBSubcategory: TlsWindbg

PRB: Excel's =REQUEST() from DDEML Application Returns #N/A

PSS ID Number: Q107980

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

When executing the =REQUEST() macro to request data from a DDEML server application, Excel returns a value of "#N/A", although the server application returned a valid data handle from the request.

CAUSE
=====

When Excel executes a =REQUEST, it requests data in the most efficient format available. Verifying through DDESPY when Excel executes the =REQUEST macro, one can see that Excel sends out a request for data for each format, in this order:

```
XLTable  (Excel fast table format)
BIFF4    (Excel 4.0 file format)
BIFF3    (Excel 3.0 file format)
SYLK     (Symbolic Link)
WK1      (Lotus 1-2-3 release 2 format)
CSV      (comma-delimited text)
TEXT     (CF_TEXT)
RTF      (rich text format)
DIF      (data interchange format)
```

Knowing what formats it can handle best, Excel requests data in the most efficient format first, and so on, until it finds one that the server application supports. At this point, Excel stops sending further requests.

In response to a request, a DDEML server application that supports only one format (for example, the CF_TEXT format) may return a data handle in CF_TEXT format, regardless of the format being requested. When Excel sends its first request for data in XLTable format, this server application returns a data handle in CF_TEXT format, as demonstrated in the code below:

```
case XTYP_REQUEST:
   if ((ghConv == hConv) &&
        (!DdeCmpStringHandles (hsz1, hszTopicName)) &&
        (!DdeCmpStringHandles (hsz2, hszItemName)))  {

          lstrcpy (szBuffer, "The Simpsons");
```

```
            return (DdeCreateDataHandle (idInst,
                                szBuffer,
                                lstrlen (szBuffer)+1,
                                0L,
                                hszItemName,
                                CF_TEXT,
                                0);
      }
      return (HDDEDATA)NULL;
```

Because Excel expected to receive data in the format it had requested (that is, XLTable format), and instead received date in CF_TEXT format, Excel returns #N/A, not knowing how to handle the data it received.

RESOLUTION
==========

In response to a request, a DDEML server application should return a valid data handle only for the format it supports. When processing an XTYP_REQUEST transaction, the server application should first check whether the data being requested is in its supported format; if so, the server application should return an appropriate data handle. Otherwise, the server application should return NULL.

The code above can be modified as follows to check for this condition:

```
   case XTYP_REQUEST:
      if ((ghConv == hConv) &&
           (!DdeCmpStringHandles (hsz1, hszTopicName)) &&
           (!DdeCmpStringHandles (hsz2, hszItemName)) &&
           (wFmt == CF_TEXT))  {     // Add this to the if clause
                                     // to check if data is being requested
                                       // in one of its supported formats.

               lstrcpy (szBuffer, "Fred Flintstone");
               return (DdeCreateDataHandle (idInst,
                                szBuffer,
                                lstrlen (szBuffer)+1,
                                0L,
                                hszItemName,
                                CF_TEXT,
                                0);
        }
        return (HDDEDATA)NULL;
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrDde

PRB: ExitProgman DDE Service Does Not Work If PROGMAN Is Shell

PSS ID Number: Q69899

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

Calling the ExitProgman() function documented in the Microsoft Windows SDK version 3.0 "Guide to Programming," section 22.4.4 (pages 22-19 through 22-22) fails under certain circumstances.

CAUSE

Calling this function will fail if the Program Manager is the Windows shell.

RESOLUTION

This behavior is by design. The Windows 3.1 documentation states:

> If Program Manager was started from another application, the ExitProgman command instructs Program Manager to exit and, optionally, save its groups information.

For another application to start Program Manager, the Program Manager cannot be the shell.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrDde

PRB: ExitWindowsEx with EWX_LOGOFF Doesn't Work Properly

PSS ID Number: Q149690

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

========

When an application calls ExitWindowsEx with EWX_LOGOFF in Windows 95, the application itself does not exit. Additionally, the use of the EWX_FORCE flag in conjunction with EWX_LOGOFF will cause undefined behavior.

CAUSE

=====

It is the application's responsibility to terminate itself. However, limitations of the implementation of ExitWindowsEx in Windows 95 require that the application execute the ExitWindowsEx call and terminate itself in a particular way.

RESOLUTION

==========

Code similar to the following will correctly perform a logoff in Windows 95:

```
   case IDM_LOGOFF:
      {
      DWORD    dw;
      HANDLE   hThread;

      g_hwndParent = hWnd;

      hThread = CreateThread( NULL,
                              0,
                              (LPTHREAD_START_ROUTINE)ShutdownThread,
                              (LPVOID)0,
                              0,
                              &dw);

      if(hThread)
         CloseHandle(hThread);
      }
      break;

/****************************************************************************

   ShutdownThread()

****************************************************************************/
```

```
BOOL WINAPI ShutdownThread(DWORD dwCmd)
{
BOOL  f;

f = ExitWindowsEx(EWX_LOGOFF, 0);

//if the shutdown worked, terminate the calling app
if(f)
   {
   if(g_hwndParent)
       {
       //shutdown the calling app ASAP, don't destroy the window
       PostMessage(g_hwndParent, WM_QUIT, 0, 0);
       }
   }

return f;
}
```

STATUS
======

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbprg kbui kbprb
KBSubcategory: UsrCtl

PRB: ExtTextOutW or TextOutW Unicode Text Output Is Blank

PSS ID Number: Q145754

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SYMPTOMS
========

When you use ExtTextOutW or TextOutW to display a character that is in the unicode range but not in the particular codepage with which the font file is associated, a blank character is displayed.

CAUSE
=====

The Unicode font glyphs are stored inside the font file. However, one font file does not contain all the glyphs required to display all 64K possible Unicode characters.

Windows NT and Windows 95 support the Unicode Font file. Therefore all of the Unicode glyphs can be rendered if the glyph is found in the font file. However, Windows 95 does not support the Unicode characters other than the characters for which the particular codepage of the font file is defined. When Windows 95 opens a font file, it will first determine which encoding standard the font file contains (for example the ANSI, SHIFTJIS, BIG5, or HANGEUL character set) by looking at the LanguageID in the name table. After Windows 95 determines the file's LanguageID, the font file can only support that particular language, even though other glyphs may exist in the font file. For example, a Japanese font file can not support Hangeul because the Japanese codepage does not define any Hangeul characters. But a Japanese font can support part of Cyrillic because the Japanese codepage defines part of those.

On the other hand, Windows 95 supports multiple character sets from one font file. For a given font file that supports multiple character sets, Windows 95 will enumerate the multiple character sets as a seperate fonts.

The user can determine if a font supports multiple character sets in the Font common dialog under the Script combo box. If the selected font supports multiple character sets, then the Script combo box will have entries for the various scripts available.

RESOLUTION
==========

The functionality as described above is by design and is different from the Windows NT Unicode Font support. In Far East Windows NT versions, FontLinks make it possible to have a font which can supports whole 64K Unicode characters. The difference between Windows 95 and Windows NT is due to the fact that the former is ANSI-based while the latter is Unicode-based.

However, Windows NT does not support a method to input all characters while Windows 95 defines those methods as multilingual specification.

In furture versions of Windows 95 and Windows NT, the multilingual specification will be merged into both platforms.

STATUS

This behavior is by design.

MORE INFORMATION

See the Win32 SDK's Multilingual support sections for more information on multiple character set handling and multilingual operation. Additionally, see Developing International Software For Windows 95 and Windows NT, Microsoft Press 1995.

Additional reference words: 4.00
KBCategory: kbprg kbprb
KBSubcategory: wintldev

PRB: FindExecutable() Truncates Result at First Space in LFN

PSS ID Number: Q140724

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:

    - Microsoft Windows 95
    - Microsoft Windows NT versions 3.1, 3.5, 3.51

SYMPTOMS
========

The FindExecutable function is supposed to retrieve the fully qualified path to the executable (.exe) file associated with the specified file name. For example, the following call to FindExecutable() should return the path to Winword.exe:

```
FindExecutable ("C:\\README.DOC", NULL, szBuffer);
```

Assuming Winword.exe is currently in the C:\Msoffice\Winword directory, the function, upon return, should fill szBuffer with the string C:\Msoffice\Winword\Winword.Exe.

However, when you use FindExecutable() on a file whose associated application is in a directory that has a long file name (LFN) that includes a space, the function truncates the string up at the first space. Going back to the previous code example, if Winword.exe is located in the C:\Program Files\My Accessories directory, the function incorrectly returns the truncated string C:\Program instead of the following expected string:

```
C:\Program Files\My Accessories\Winword.Exe
```

CAUSE
=====

File associations are stored in the registry under HKEY_CLASS_ROOT, where the executable name is actually stored under the key

```
HKEY_CLASSES_ROOT\<file Type>\shell\open\command = <Name of Executable>
```

By design, long file names stored in the registry should be enclosed in quotation marks. Otherwise, the system is made to treat the rest of the characters following the space as arguments.

In the example, the path to WinWord.Exe should be stored in the registry as:

```
HKEY_CLASSES_ROOT\Word.Document.6\shell\open\command =
 "C:\Program Files\My Accessories\Winword.Exe" /n
```

which does causes FindExecutable to return the expected string.

Note that this problem does not occur when the short path name is stored in the registry, instead of the long file name. Again, taking the same example to WinWord.exe, this should not be a problem if the short path name "C:\Progra~1\MYACCE~1\WINWORD.EXE" is stored in the registry instead of the LFN version.

WORKAROUND
==========

In some situations, applications do not have control over what gets stored in the registry. It may be the associated application's setup program that wrote the incorrect LFN string (without quotation marks) to the registry. Any application that would then call FindExecutable() on this associated application will run into this problem.

One workaround to this problem might be to parse the string returned and replace the \0 character with a space. Stepping through a debugger, after FindExecutable() returns, you will quickly find that although the space has been replaced with the NULL character (\0), the rest of the string is left intact, so simply reverting the \0 character back to a space gives the expected string.

STATUS
======

This behavior is by design.

Additional reference words: 4.00 3.10 3.50 invalid wrong
KBCategory: kbprg kbprb
KBSubcategory: UsrShell

PRB: FindText, ReplaceText Hook Function

PSS ID Number: Q96135

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

When adding a hook function to the FindText and/or ReplaceText common dialog box, the dialog box is not shown. However, the hook function is receiving messages and the dialog box window does exist.

CAUSE
=====

The hook function is returning FALSE after processing the WM_INITDIALOG message.

RESOLUTION
==========

After processing the WM_INITDIALOG message in the hook function, return TRUE. If your hook function is setting the focus to a specific control, and therefore should return FALSE, add the following code:

```
   case WM_INITDIALOG:

      ...WM_INITDIALOG code...

      SetFocus(hCtrl);
      ShowWindow(hDlg, SW_NORMAL);
      UpdateWindow(hDlg);
      return(FALSE);
```

This code ensures that the dialog box is visible.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrCmnDlg

PRB: GetExitCodeProcess() Always Returns 0 for 16-Bit Processes

PSS ID Number: Q111559

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5
    - Microsoft Win32s versions 1.1, 1.15, and 1.2

SYMPTOMS

GetExitCodeProcess() always returns a status of 0 (zero) when the handle for a 16-bit process is passed. This applies to both Windows NT and Win32s.

STATUS

This behavior is by design in the Microsoft products listed at the beginning of this article. Microsoft may add functionality in future versions that support exit codes from 16-bit processes.

Additional reference words: 3.10 3.50 1.10 1.20
KBCategory: kbprg kbprb
KBSubcategory: BseMisc W32s

PRB: GetLogicalDrives() Indicates that Drive B: Is Present

PSS ID Number: Q126626

The information in this article applies to:

- Microsoft Win32s, versions 1.2 and 1.25

SYMPTOMS

GetLogicalDrives() returns a bitmask that indicates that drive B is an available drive present on the system, even though there is no physical drive B.

CAUSE

Drive B is a ghosted drive, so you can use it even if it does not exist. This is useful for performing a diskcopy.

RESOLUTION

Use GetDriveType() to determine whether drive B: is present as a physical device.

STATUS

This behavior is by design.

Additional reference words: 1.20 1.25
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: GetOpenFileName nMaxFile Interpreted Incorrectly

PSS ID Number: Q137194

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SYMPTOMS

In a 16-bit Windows-based application, call GetOpenFileName with a string buffer of 10 characters and set nMaxFile to 10. Double-click a file whose full path name is 10 characters long. The file name is returned and the 11th character in the buffer is set to 0. This is a problem because the application has written beyond the specified length of the buffer.

CAUSE

Windows 3.1 had this same problem, so this behavior was maintained in Windows 95 for compatibility reasons.

RESOLUTION

Applications should make sure that they can handle having this API overwrite one more byte than the size that they passed in.

STATUS

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbui kbprb
KBSubcategory: UsrCmnDlg

PRB: GetOpenFileName() and Spaces in Long Filenames

PSS ID Number: Q108233

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS

GetOpenFileName() is the application programming interface (API) for the open file common dialog box. This API displays the long filenames (LFNs) on NTFS and HPFS.

When using the OFN_ALLOWMULTISELECT flag with the GetOpenFileName() API, the dialog box automatically presents the 8.3 names for all LFNs that contain embedded spaces.

CAUSE

The original design of GetOpenFileName() uses a filename list that is space-delimited when the OFN_ALLOWMULTISELECT flag is specified. Thus, there is no programmatic way to determine which string tokens are complete filenames or fragments of a complete name with spaces.

STATUS

This behavior is by design. Microsoft is considering changing this behavior in a future release of Windows NT.

MORE INFORMATION

Historically, FAT filenames that contained embedded spaces were branded as "illegal," even though the specifications of the FAT file system do not impose such a restriction. For example, many of the MS-DOS command-line utilities do no allow the user to specify filenames with embedded spaces, because of difficulties that would be introduced in parsing the command line. Under Windows NT, the command utilities have been enhanced to support such names if they are in quotation marks.

Additional reference words: 3.10 3.50
KBCategory: kbui kbprb
KBSubcategory: UsrCmnDlg

PRB: Getsockopt() Returns IP Address 0.0.0.0 for UDP

PSS ID Number: Q129065

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.11 and 4.0
- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SYMPTOMS

By following the steps listed below, you might think you should get back the interface address over which the connection was made. However, it actually returns the address 0.0.0.0.

1. Open a UDP socket.

2. Bind it to INADDR_ANY.

3. Call connect() to make a UDP connection.

4. Call getsockname() on your socket.

However, if it was a TCP socket, you would get back the IP address of the interface.

CAUSE

UDP

This is the behaviour expected from some flavors of UNIX, notably those derived from BSD. When an application calls connect() on a UDP socket that is bound to INADDR_ANY, the operating system associates the remote address with the local socket. This saves the programmer from having to specify the remote IP address in each sendto() or recvfrom(). Instead they may use send() and recv(). Note that this is just a convenience provided by the operating system; there is no network traffic associated with this call. At this point, the underlying IP software determines the interface over which packets will be sent. As described earlier, under BSD UNIX, calling getsockname() will return the IP address of the interface to the application.

This however, is not expected behaviour under Windows NT, Windows 95, or Microsoft TCP IP/32 for Windows for Workgroups version 3.11. Calling getsockname() will return the IP address 0.0.0.0 (INADDR_ANY). Applications should not assume that they can get the IP address of the interface.

TCP

The behaviour is different if it was a TCP socket. In this case, calling getsockname() on a connected socket that was bound to INADDR_ANY will return the IP address of the interface over which the connection was made. The state of the connection can also be observed by typing 'netstat' at a command prompt.

NOTE: To enumerate all the IP addresses on an IP host, the application should call gethostname(), call gethostbyname(), and then iterate through the h_addr_list[] member of the hostent struct returned by gethostbyname() as in this example:

```
   char     Hostname[100];
   HOSTENT *pHostEnt;
   int      nAdapter = 0;

   gethostname( Hostname, sizeof( Hostname ));
   pHostEnt = gethostbyname( Hostname );

   while ( pHostEnt->h_addr_list[nAdapter] )
   {
      // pHostEnt->h_addr_list[nAdapter] -the current address in host order
      nAdapter++;
   }
```

STATUS
======

This behavior is by design.

Additional reference words: 3.11 4.00 3.10 3.50 3.51
KBCategory: kbnetwork kbprb
KBSubcategory: NtwkWinsock

PRB: GetVolumeInformation() Fails with UNC Name

PSS ID Number: Q119219

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

GetVolumeInformation() fails and GetLastError() returns 123 (ERROR_INVALID_NAME) if a UNC name is used. The UNC name has the form \\<SERVER>\<SHARE>.

RESOLUTION
==========

GetVolumeInformation() requires an extra backslash with UNC names, so that the name has the form \\<SERVER>\<SHARE>\.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg kbprb
KBSubcategory: BseMisc

PRB: GlobalAlloc() Pagelocks Blocks on Win32s

PSS ID Number: Q114611

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, and 1.2

SYMPTOMS
========

If a Win32-based application running in Win32s uses GlobalAlloc() to allocate memory from the global heap with GMEM_FIXED, with GPTR, with GMEM_ZEROINIT, or without specifying GMEM_MOVEABLE the memory allocated will be fixed and page-locked.

CAUSE
=====

When a Win32 application running under Win32s on Windows 3.1 calls GlobalAlloc() the call is translated via a thunk supplied by Win32s in a 16-bit DLL. The 16-bit DLL then calls the Windows 3.1 function GlobalAlloc(). When GlobalAlloc() is called from a DLL in Windows 3.1 the allocated memory will be fixed and page-locked unless GMEM_MOVEABLE is specified.

RESOLUTION
==========

The GlobalAlloc() flags should always include GMEM_MOVEABLE if memory does not need to be fixed and page-locked. This is expected behavior for Windows 3.1.

MORE INFORMATION
================

A Windows-based application will not fix or page-lock memory even when specifically using the GMEM_FIXED flag. This behavior is unique to Windows version 3.1; using GlobalAlloc() with GMEM_FIXED to allocate fixed and page-locked memory must be done in a DLL.

In Windows 3.1, the GMEM_FIXED flag is defined as 0x0000. Using GMEM_ZEROINIT without GMEM_MOVEABLE will command GlobalAlloc() to allocate using GMEM_FIXED by default. Since Win32s passes all GlobalAlloc() calls to the Windows 3.1 GlobalAlloc() by a DLL, GlobalAlloc() called from either a Win32 application or a Win32 DLL will allocate the block fixed and page-locked unless the GMEM_MOVEABLE flag is specified.

The following code illustrates this case:

```
{
HGLOBAL hMem;

// allocate a block from the global heap
```

```
   hMem = GlobalAlloc(GMEM_ZEROINIT, 512);

     .
     .
     .

   }
```

Although this source code is compatible between applications for Windows 3.1 and applications for Windows NT running on Win32s, the result is different. A 16-bit application running on Windows 3.1 will allocate the memory as moveable and zero the contents. A Win32 application running on Win32s will allocate the memory as fixed and page-locked and zero the contents.

REFERENCES
==========

Appendix B, titled "System Limits", of the "Win32s Programmer's Reference Manual" briefly mentions on page 56 not to use GMEM_FIXED in GlobalAlloc() called by 32-bit applications.

Additional reference words: 1.10 1.20 3.10
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: GP Fault Caused by GROWSTUB in POINTER.DLL

PSS ID Number: Q117864

The information in this article applies to:

- Microsoft Win32s versions 1.1, 1.15, and 1.2

SYMPTOMS

As soon as your Win32-based application starts, there is a GP fault caused by GROWSTUB in POINTER.DLL.

CAUSE

This problem is caused by a bug in GROWSTUB, which is part of the Microsoft Mouse driver version 9.01.

RESOLUTION

Microsoft Mouse driver version 9.01b corrects this problem. Driver 9.01b is available via part number: 135-099-351 "Intellepoint Mouse Driver 2.0(dual)". This driver does not introduce new functionality, therefore, you need only upgrade if you have run into this problem.

For more information, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q119775
   TITLE   : HD1061: POINTER.DLL Corrects GP Fault with Win32 Apps

You can also avoid the problem by removing POINTER.EXE from the load= line in your WIN.INI file.

Additional reference words: 1.10
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: GP Fault in DDEML from XTYP_EXECUTE Timeout Value

PSS ID Number: Q83999

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

A general protection (GP) fault occurs in DDEML When the following occurs:

1. A DDEML (Dynamic Data Exchange Server Library) server application requires more time to process a XTYP_EXECUTE transaction than the timeout value specified by a DDEML client application
2. The server application creates windows as part of its processing
3. The client application abandons the transaction because the transaction timed out

CAUSE

The server application receives a window handle with the same value as the hidden window created to control the transaction.

RESOLUTION

Specify a timeout value in the client application longer than the time required by the server application to complete the task.

MORE INFORMATION

To use DDEML, an application (either a client or a server) registers a callback function with the library. The DDEML calls the callback function for any DDE activity. A DDE transaction is similar to a message; it contains a named constant, accompanied by other parameters.

A client application issues a XTYP_EXECUTE transaction to instruct the server application to execute a command. When a client calls the DdeClientTransaction function to issue a transaction, it can specify a timeout value, which is the amount of time (in seconds) the client is willing to wait while the server processes the transaction. If the

server fails to execute the command within the specified timeout value, the DDEML sends a message to the client that the transaction timed out. Upon receipt of this message, the client can inform the user, reissue the command, abandon the transaction, or take other appropriate actions.

If a client application specifies a short timeout period (one second, for example) and the server requires fifteen seconds to execute a command, the client will receive notification that the transaction timed out. If the client terminates the transaction, which is an appropriate action, the DDEML will GP fault.

When the client sends an XTYP_EXECUTE transaction, the DDEML creates a hidden window for the conversation. If the client calls the DdeAbandonTransaction function to terminate the transaction, the DDEML destroys the associated hidden window.

At the same time, the server application processes the execute transaction, which might involve creating one or more windows. If the server creates a window immediately after the DDEML destroys a window, the server receives a window handle with the same value as that of the destroyed window. After the server completes processing the execute transaction, it returns control to the DDEML.

Normally, the DDEML determines that the callback function is returning to a conversation that has been terminated. It calls the IsWindow function with the window handle for the transaction's hidden window to ensure that the handle remains valid.

Because the window handle has been allocated to the server application, the IsWindow test succeeds. However, this handle no longer corresponds to the transaction's hidden window. Therefore, when the DDEML attempts to retrieve the pointer kept in the hidden window's window extra bytes, the pointer is not available. When the DDEML uses the contents of this memory, a GP fault is likely to result.

The current way to work around this problem is to specify a timeout value in the client application that is longer than the time required by the server to complete its processing.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrDde

PRB: GPF When Spawn Windows-Based App w/ WinExec() in Win32s

PSS ID Number: Q121095

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, and 1.2

SYMPTOMS

Win32-based applications running under Win32s can spawn both Windows-based and Win32-based applications by using either WinExec() or CreateProcess(). However, there is a case where spawning a Windows-based application with WinExec() does not work as expected and may cause a general protection (GP) fault.

CAUSE

There is a bug in the C start-up code that comes with Microsoft C version 6.0. If you spawn an application built with Microsoft C version 6.0 by calling LoadModule() with an explicit environment, the application does not run correctly. This is true whether the application was spawned from a Win32-based application or a Windows-based application. Win32s calls LoadModule() with an explicit environment when you spawn a Windows-based application with WinExec(). As a result, under Win32s version 1.1 and 1.15, WinExec() will report success, but the Windows-based application built with Microsoft C version 6.0 may cause a GP fault.

RESOLUTION

The best solution is to rebuild the application with another compiler package. However, because a number of Windows accessories (such as Notepad and Write) were built with Microsoft C version 6.0 and you cannot modify these applications, changes were introduced into Win32s version 1.2 to help you work around this problem. These changes are detailed in the More Information section below.

NOTE: Win32s uses a different mechanism to spawn Win32-based applications, so the problems discussed in this article do not occur when spawning Win32-based applications with WinExec().

MORE INFORMATION

In Win32s version 1.2, WinExec() does not pass the environment to the spawned application (child). The child receives the standard global environment strings. This allows the application to run, but the child does not receive the modified environment from the parent. This seemed to be a reasonable compromise, because most applications do not change the environment for the child. If an application must modify the child's environment, it can spawn the application using CreateProcess() and specify

an explicit environment. However, if the child was built using Microsoft C version 6.0, it may cause a GP fault. In addition, if the parent exits, the child's environment becomes invalid. These three problems are not specific to Win32s and will happen with Windows-based applications as well.

Additional reference words: 1.00 1.15 1.20 GPF
KBCategory: kbprg kbprb
KBSubCategory: W32s

PRB: Help Error: Unable To Display the Find Tab (177)

PSS ID Number: Q142222

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

In Winhlp32.exe, you may encounter the following error when you click the Find tab in the Help Topics dialog box:

   Unable to display the Find tab. (177)

This error may occur immediately upon clicking the Find tab, or it may occur just after the word list has been built (during the first time the help file is run).

CAUSE

The cause of this error depends on when it occurs. If it occurs immediately upon clicking the Find tab, the problem may be an old or corrupted .gid file. Normally, when help is started for the first time, it builds the Full Text Search database (.fts) and the index file (.gid). However, if a .gid file already exists in the current directory, help may become confused and display the error message when the Find tab is clicked.

If the error message occurs immediately after the word list has been built, the problem is most likely due to help attempting to index a help file that contains no keywords. This can happen if a helpfile is referenced by way of an Index command in a contents file (.cnt) and that helpfile has no K footnotes.

RESOLUTION

If the problem is due to a damaged or corrupted .gid file, delete the existing .gid and .fts files and re-launch the help file. Note that the .gid file is a hidden file that resides in the same directory as the help file.

If the problem is due to indexing a help file that contains no keywords, either remove the help file from the index entirely or use the Link command in the contents (.cnt) file to reference the help file. If you are unsure whether or not a help file has keywords in it, you can either search the .rtf files for K footnotes, or you can use the reporting mechanism in Help Workshop. To do the latter, click Report on the File menu, specify the help file to search, specify a file to output the results to, and choose K

keywords for Display options. This gives you a list of all of the keywords
in the specified helpfile.

Additional reference words: 4.00
KBCategory: kbtool kbprb
KBSubcategory: tlshlp

PRB: Icon Handlers in Start Menu Don't Match Those in Explorer

PSS ID Number: Q142276

The information in this article applies to:

- Microsoft Win32 Software Development Kit for
  Microsoft Windows 95

SYMPTOMS

Icons shown on items in the start menu may not match those shown in Windows Explorer for the same items if an icon handler is being used to generate the icons and the GIL_DONTCACHE flag is present.

RESOLUTION

There is a problem in the Start menu code handling the GIL_DONTCACHE flag. If that flag is removed from the code, the start menu will work correctly. If the GIL_DONTCACHE is being used to ensure that the right icons are geting cached, you can solve that same problem by using SHChangeNotify(SHCNE_UPDATEDIR, SHCNF_PATH, ...) to notify the shell whenever the handler code makes a change to the icons.

STATUS

This behavior is by design.

Additional reference words: 4.00 shell extension icon handler
KBCategory: kbui kbprb
KBSubcategory:

PRB: Inadequate Buffer Length Causes Strange Problems in DDEML

PSS ID Number: Q107387

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

Specifying an inadequate buffer length for an XTYP_POKE or an XTYP_EXECUTE command causes strange problems in DDEML.

Problems can range from a general protection (GP) fault or Exception 13, to DDEML timeout errors (such as DMLERR_EXECACKTIMEOUT or DMLERR_POKEACKTIMEOUT) or a DDEML transaction failure (or DMLERR_NOTPROCESSED). Sometimes, the application may seem to work for the most part, and then occasionally crash.

Data can be passed to the server application via XTYP_POKE or XTYP_EXECUTE in two ways:

- Directly, as a pointer to the data or command string, as in the sample code below:

```
char lpszString [80];

lstrcpy (lpszString, "[FileOpen(""C:\README.DOC"")]");
DdeClientTransaction (lpszString,             // string buffer
                      lstrlen (lpszString)+1, // string buffer length
                      hConv,
                      hszItem,
                      CF_TEXT,
                      XTYP_POKE,
                      1000,
                      NULL);
```

-or-

- By creating a data handle, and passing that on to the DdeClientTransaction() call:

```
char lpszString [80];
HDDEDATA hData;

lstrcpy (lpszString, "[FileOpen(""C:\README.DOC"")]");
hData = DdeCreateDataHandle (idInst,
                             lpszString,
```

```
                                lstrlen (lpszString)+1,
                                0,
                                NULL,
                                CF_TEXT,
                                0);
   if (!hData)
      DdeClientTransaction (hData,     // string buffer
                            -1,        // indicates hData is a data handle
                            hConv,
                            hszItem,
                            CF_TEXT,
                            XTYP_POKE,
                            1000,
                            NULL);
```

CAUSE
=====

Because data is most commonly passed between applications in CF_TEXT format, a common problem with the string buffer length is setting it to lstrlen (lpszString), where lpszString is the buffer containing the string the client needs to pass to the server. Because the lstrlen() function does not include the terminating null character, this can cause the system to append garbage characters to the end of the string, thus sending an invalid string to the server application.

RESOLUTION
==========

When passing strings between two applications, the string buffer length should be set to lstrlen (lpszString) +1, to include the terminating null character ('\0').

Using DDESPY, it is easy to track down this problem, because one can follow the string being passed from the client to the server application. Garbage characters incorrectly being appended to the string usually indicate a problem with specifying an inadequate string buffer length.

Additional reference words: 3.10 3.50 3.51 4.00 95 gpf gp-fault
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrDde

PRB: Inconsistencies in GDI APIs Between Win32s and Windows NT

PSS ID Number: Q123421

The information in this article applies to:

- Microsoft Win32s version 1.2

SYMPTOMS

The StretchBlt() and StretchDIBits()/SetDIBits() GDI APIs do not behave consistently under Win32s and Windows NT.

StretchBlt()

If the source width and height specified in the call to StretchBlt() are greater than the actual bitmap width and height, StretchBlt() fails. The same call to StretchBlt() succeeds under Windows NT.

StretchDIBits()/SetDIBits()

If the memory pointed to by the lpBits parameter is read-only, the call to StretchDIBits()/SetDIBits() fails.

NOTE: When a Win32-based application uses the memory returned from LockResource() as a parameter to SetDIBits(), by default, it's using read-only memory, because the resource section is defined by default as read-only.

CAUSE

These problems are due to bugs in Windows. In the case of StretchDIBits() and SetDIBits(), Windows mistakenly verifies that the buffer is writable. This problem does not show up in a 16-bit Windows-based application running under Windows because resources are loaded into read/write (global) memory.

RESOLUTION

In Win32s version 1.25, Win32s will always make the resource section read/write, regardless of what is specified in the section attributes. This will work around the problem. In the meantime, use the following resolutions:

StretchBlt()

To work around the problem, specify the proper width and height for the source bitmap.

StretchDIBits()/SetDIBits()

To work around the problem, do one of the following:

- Copy the memory to a temporary read/write buffer.

  -or-

- Use the linker switch /SECTION:.rsrc,rw to make the resource section read/write. Windows NT will allocate separate resource sections for each copy of the application.

Additional reference words: 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: InstallSHIELD Error: Setup Requires a Different Version

PSS ID Number: Q140617

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) for:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

After installing InstallSHIELD SDK Edition from Microsoft Visual C++ version 4.0, you may see the following error message when you try to run the Template application:

   Setup requires a different version of Windows. Check to
   make sure that you are running Setup on the Windows platform
   for which it is intended.

   Error 102.

CAUSE
=====

The message occurs when a the _inst32i.ex_ does not exist in the directory where Setup.exe is located.

RESOLUTION
==========

To correct the problem in both the Template application and in projects created with InstallSHIELD SDK Edition, copy _inst32i.ex_ from the \Ishield\Disk1 directory on the Visual C++ compact disc into the following two locations on the hard drive:

   \Program Files\Stirling\InstallShield\Program

   -and-

   \Program Files\Stirling\InstallShield\Template\One\Disk1

STATUS
======

This behavior is by design.

MORE INFORMATION
================

This behavior occurs only when InstallSHIELD is installed from Microsoft Visual C++ version 4.0. It does not occur when InstallSHIELD is installed

from the Microsoft Win32 SDK.

Additional reference words: 4.00
KBCategory: kbsetup kbprb
KBSubcategory: ishield

PRB: Inter-thread SetWindowText() Fails to Update Window Text

PSS ID Number: Q125687

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

Calling SetWindowText() to set a static control text from a thread other than the one that created the control fails to display the new text in Windows 95.

CAUSE
=====

When SetWindowText() is called from another thread, instead of sending a WM_SETTEXT message to the appropriate window procedure, only DefWindowProc() is called, so the edit and static controls do not paint the control appropriately because the appropriate code is never executed, so the text on the screen is never updated. In other words, calling SetWindowText() updates the buffer internally, but the change is not reflected on the screen.

RESOLUTION
==========

One obvious workaround is to refrain from calling SetWindowText() from another thread, if possible.

If design considerations don't allow doing this, use one of these workarounds:

- Send a WM_SETTEXT message directly to the window or control.

  -or-

- Call InvalidateRect() immediately after the SetWindowText(). This works because DefWindowProc() updates the buffer where the text is stored.

STATUS
======

This inter-thread SetWindowText() behavior is by design in Windows version 3.x. It was maintained in Windows 95 for backward compatibility purposes. Applications written for Windows version 3.x can expect their inter-thread SetWindowText() calls to behave as they did in Windows version 3.x.

MORE INFORMATION
================

Calling SetWindowText() from another thread in Windows NT displays the window text correctly, so it works differently from Windows 95.

Additional reference words: 4.00
KBCategory: kbui kbprb
KBSubcategory: UsrWndw

PRB: IsCharAlpha Return Value Different Between Versions

PSS ID Number: Q84843

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS

Under Windows version 3.1, the IsCharAlpha function returns TRUE for the character values 8Ah, 8Ch, 9Ah, 9Ch, 9Fh, and DFh. Under Windows version 3.0, the function returns FALSE for these character values.

CAUSE

These characters represent alphabetic characters that were added to the Windows character set in Windows 3.1.

RESOLUTION

Applications that use the IsCharAlpha function should behave properly with the newly-defined characters. No changes should be required.

MORE INFORMATION

Appendix C.1, page 596, of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 3: Messages, Structures, and Macros" lists the Windows character set.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrLoc

PRB: IsGdiObject() Is Not a Part of the Win32 API

PSS ID Number: Q91072

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

There is no IsGdiObject() function in the Win32 API.

CAUSE

The function was added to the Windows 3.1 API because passing a handle to a non-GDI object to a GDI function causes a GP fault under Windows 3.0. Windows NT and Windows 95 detect whether the APIs are passed an inappropriate handle, so the function can return an error.

RESOLUTION

IsGdiObject() is not needed on Windows NT or Windows 95.

Additional reference words: 3.10 3.50
KBCategory: kbgraphic kbprb
KBSubcategory: GdiMisc

PRB: JournalPlayback Hook Can Cause Windows NT to Hang

PSS ID Number: Q124835

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS

Incorrect use of the delay return value from a journal playback hook can cause Windows NT to hang temporarily.

CAUSE

The menu loop in Windows NT calls PeekMessage() with the PM_NOREMOVE flag, does some processing, and then removes the message from the queue. This sequence is repeated until the menu goes away. When JournalPlayback is occurring, the PeekMessage(PM_NOREMOVE) results in a callback to the application's JournalPlaybackProc with an HC_ACTION code. The subsequent PeekMessage(PM_REMOVE) also calls the JournalPlaybackProc with an HC_ACTION code. If the peek is successful, it is followed by an HC_SKIP callback.

In order to have playback from a journal playback hook occur at a certain rate, Microsoft designed it so that the value returned by the JournalPlaybackProc can be non-zero. This value represents the number of clock-ticks the system should wait before processing the event. What the documentation doesn't make clear is that when the delay has expired, another callback to the JournalPlaybackProc is made to obtain the same event again; the event provided with the previous non-zero delay is not used at all. All subsequent HC_ACTION calls that request the same event should be returned with a zero delay value. Only after an HC_SKIP callback has been made, may an HC_ACTION callback return a non-zero delay value again. Some applications do not do this correctly, and simply alternate between returning a delay and returning a non-delay.

This alternating delay/no delay method made the Windows NT menu loop hang because the PeekMessage(PM_NOREMOVE) would get an input event (with no delay), then the PeekMessage(PM_REMOVE) would get a non-zero return value from the JournalPlaybackProc. This represents no message -- so instead of issuing an HC_SKIP callback to the JournalPlaybackProc to advance to the next event, the Windows NT menu loop code simply looped back to the PeekMessage(PM_NOREMOVE) getting stuck in an infinite loop.

RESOLUTION

To work around this problem, make sure the JournalPlaybackProc correctly returns the delay only for the first request for an event.

Neither Windows version 3.1 nor Windows 95 have this problem.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

The following sample code demonstrates correct and incorrect methods of handling delays in a journal playback hook.

Sample Code

```
LRESULT CALLBACK JournalPlaybackProc(
    int nCode,
    WPARAM wParam,
    LPARAM lParam)
{
    static BOOL     fDelay;
    static EVENTMSG event;
    static LRESULT  ticks_delay;
    BOOL            fCallNextHook = FALSE;
    LRESULT         lResult = 0;

    switch( nCode )
    {
        case HC_SKIP:
            fDelay = TRUE;      // <<<< CORRECT PLACE TO RESET fDelay

            // Get the next event from the list.  If the routine returns
            // FALSE, then we are done - release the hook.
            if( !GetNextEvent( &event, &ticks_delay ))
                SetJournalHook( FALSE, NULL );
            break;

        case HC_GETNEXT:
        {
            // Structure information returned from previous GetNextEvent
            // call
            LPEVENTMSG lpEvent = (LPEVENTMSG) lParam;

            // Set the event
            *lpEvent = event;

            if( fDelay )
            {
                // Toggle pause variable so that the next call won't
                // pause.  Return the pause length specified by ticks_delay
                // since this is the first time the event has been
                // requested.
```

```
                fDelay = FALSE; // <<<< CORRECT PLACE TO CLEAR fDelay
                return( ticks_delay );
            }
            break;
        }

        case HC_SYSMODALOFF:
            // System modal dialog is going away - something really got
            // hosed.  Windows took care of removing our JournalPlayback
            // hook, so no need to call SetJournalHook( FALSE ).

            fCallNextHook = TRUE;
            break;

        case HC_SYSMODALON:
        default:
            // Something is is not right here, let the next hook handle
            // it.

            fCallNextHook = TRUE;
            break;
    }

    // If the event wasn't processed by our code, call next hook
    if( fCallNextHook )
        lResult = CallNextHookEx( s_journalHook, nCode, wParam, lParam );

    // fDelay = TRUE;           // <<<< WRONG PLACE TO RESET bDelay !!!
    return lResult;
}
```

Additional reference words: 3.10 3.50
KBCategory: kbui kbprb
KBSubcategory: UsrHks

PRB: Large DIBs May Not Display Under Win32s

PSS ID Number: Q126575

The information in this article applies to:

- Microsoft Win32s versions 1.10, 1.15, or 1.20

SYMPTOMS

DIB functions fail when using large DIBs under Win32s.

CAUSE

There is a two-megabyte limit on the size of the area of a DIB that can be blitted using blting functions under Win32s. In versions of Win32s up to 1.2, Microsoft set this size to accommodate DIB blts of 1024*768*24 bits-per-pixel. In version 1.25, the maximum size of the blitted area will be enlarged to accommodate 1280*1024*24 bits-per-pixel.

The following functions are affected:

- SetDIBits
- SetDIBitsToDevice
- CreateDIBitmap
- StretchDIBits

WORKAROUND

To work around the problem, break down large blts into bands that are smaller than two megabytes. Please keep in mind that the biSizeImage field of the BITMAPINFOHEADER used with the blting functions will need to be set to a value smaller than the DIB size limit.

The following code demonstrates a simple implementation of StretchDIBits() that can be used with large DIBs under Win32s.

```
/* Macro to determine the bytes in a DWORD aligned DIB scanline */
#define BYTESPERLINE(Width, BPP) ((WORD)((((DWORD)(Width) *
(DWORD)(BPP) + 31) >> 5)) << 2)

int NewStretchDIBits(
    HDC  hdc,     // handle of device context
    int  XDest,     // x-coordinate of upper-left corner of dest. rect.
    int  YDest,     // y-coordinate of upper-left corner of dest. rect.
    int  nDestWidth,  // width of destination rectangle
    int  nDestHeight, // height of destination rectangle
    int  XSrc,     // x-coordinate of upper-left corner of source rect.
    int  YSrc,     // y-coordinate of upper-left corner of source rect.
    int  nSrcWidth,   // width of source rectangle
    int  nSrcHeight,  // height of source rectangle
```

```
    VOID  *lpBits,    // address of bitmap bits
    BITMAPINFO *lpBitsInfo,   // address of bitmap data
    UINT  iUsage,  // usage
    DWORD  dwRop   // raster operation code
   )
{
    BITMAPINFOHEADER  bmiTemp;
    float fDestYDelta;
    LPBYTE lpNewBits;
    int i;

    // Check for NULL pointers and return error
    if (lpBits == NULL) return 0;
    if (lpBitsInfo == NULL) return 0;

    // Get increment value for Y axis of destination
    fDestYDelta = (float)nDestHeight / (float)nSrcHeight;

    // Make backup copy of BITMAPINFOHEADER
    bmiTemp = lpBitsInfo->bmiHeader;

   // Adjust image sizes for one scan line
    lpBitsInfo->bmiHeader.biSizeImage =
                      BYTESPERLINE(lpBitsInfo->bmiHeader.biWidth,
                                   lpBitsInfo->bmiHeader.biBitCount);
    lpBitsInfo->bmiHeader.biHeight = 1;

    // Initialize pointer to the image data
    lpNewBits = (LPBYTE)lpBits;

    // Do the stretching
    for (i = 0; i < nSrcHeight; i++)
   if (!StretchDIBits(hdc,
          XDest, YDest + (int)floor(fDestYDelta * (nSrcHeight - (i+1))),
          nDestWidth, (int)ceil(fDestYDelta),
          XSrc, 0,
          nSrcWidth, 1,
          lpNewBits, lpBitsInfo,
          iUsage, SRCCOPY))
        break; // Error!
   else
     // Increment image pointer by one scan line
     lpNewBits += lpBitsInfo->bmiHeader.biSizeImage;

    // Restore BITMAPINFOHEADER
    lpBitsInfo->bmiHeader = bmiTemp;

    return(i);
}
```

STATUS
======

This behavior is by design.

Additional reference words: 1.15 1.20 1.10
KBCategory: kbprg kbprb kbcode
KBSubcategory: W32s

PRB: LB_DIR with Long Filenames Returns LB_ERR in Windows 95

PSS ID Number: Q131286

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS

Sending an LB_DIR message to a list box that specifies a long filename in the lParam returns LB_ERR in Windows 95 but works fine in Windows NT version 3.51.

CAUSE

The implementation of list boxes in Windows 95 thunks down to 16-bit USER.EXE, and the LB_DIR command has not been enhanced to support long filenames.

RESOLUTION

Convert the long filename to its short form before passing it as the lParam to LB_DIR by using GetShortPathName(). Similarly, when calling DlgDirList() to fill a list box with filenames, make sure the lpPathSpec parameter refers to the short name of the file.

Sample Code

```
char  szLong [256], szShort [256];
DWORD dwResult;
LONG  lResult;

lstrcpy (szLong, "C:\\This Is A Test Subdirectory");
dwResult = GetShortPathName (szLong, szShort, 256);
if (!dwResult)
   dwResult = GetLastError ();

lstrcat (szShort, "\\*.*");
lResult = SendDlgItemMessage (hdlg,
                              IDC_LIST1,
                              LB_DIR,
                              (WPARAM)(DDL_READWRITE),
          (LPARAM)(LPSTR)szShort);
if (LB_ERR == lResult)
   // an error occurred
```

NOTE: If a file with a long filename exists under the subdirectory specified, Windows 95 displays the short name in the list box, whereas Windows NT displays the long name.

STATUS

This behavior is by design.

MORE INFORMATION

This is not a problem under Windows NT because it always supported long filenames.

You can have an application check the system version and decide at run time if it should call GetShortPathName before passing the filename as lParam to the LB_DIR message. Windows NT will, however, take a short name and fill the list box with the filenames.

Additional reference words: 4.00 1.30 LongFileName LFN DlgDirList CB_DIR DlgDirListComboBox
KBCategory: kbui kbcode kbpb
KBSubcategory: UsrCtl

PRB: Listview Comes Up with No Images

PSS ID Number: Q125628

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS

A list view is displayed with text and column headings, but the icons are not displayed.

CAUSE

The CImageList used to store the images for the list view is no longer in scope.

RESOLUTION

This can occur, for example, if you create a CImageList on the stack and create your listview, but at the point the listview is displayed, the image list has been destroyed. The ImageList functions will still return success, but no images will be displayed.

To avoid the problem, make sure your image list stays in scope.

STATUS

This behavior is by design.

Additional reference words: 4.00 Windows 95
KBCategory: kbui
KBSubcategory: UsrCtl

PRB: ListView with LVS_NOSCROLL Won't Display Header

PSS ID Number: Q137520

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0
- Microsoft Win32s version 1.3

SYMPTOMS
========

In report view, the header control is not displayed for a ListView control created with the LVS_NOSCROLL style.

CAUSE
=====

The ListView control positions the header control when the scrolling is updated. When the LVS_NOSCROLL style is specified, the control is never scrolled, so the header control is not positioned.

RESOLUTION
==========

Call following function at the appropriate time to position the header control properly. To use the function, create the ListView without the LVS_NOSCROLL style, and then call this function whenever the ListView is created, resized, the view is changed, or the parent window receives the WM_SYSPARAMETERCHANGE message. Creating the control without the LVS_NOSCROLL style will ensure that the first item in the list won't be obscured by the header control. The function will automatically detect which view is currently set and act appropriately.

```
/************************************************************************

   PositionHeader

   Call this function when the ListView is created, resized, the
   view is changed, or a WM_SYSPARAMETERCHANGE message is recieved

************************************************************************/

void PositionHeader(HWND hwndListView)
{
HWND  hwndHeader = GetWindow(hwndListView, GW_CHILD);
DWORD dwStyle = GetWindowLong(hwndListView, GWL_STYLE);

/*
   To ensure that the first item will be visible, create the control
   without the LVS_NOSCROLL style and then add it here.
*/
dwStyle |= LVS_NOSCROLL;
```

```
SetWindowLong(hwndListView, GWL_STYLE, dwStyle);

/*
   Only do this if the ListView is in report view and you were able to
   get the header hWnd.
*/
if(((dwStyle & LVS_TYPEMASK) == LVS_REPORT) && hwndHeader)
   {
   RECT        rc;
   HD_LAYOUT   hdLayout;
   WINDOWPOS   wpos;

   GetClientRect(hwndListView, &rc);
   hdLayout.prc = &rc;
   hdLayout.pwpos = &wpos;

   Header_Layout(hwndHeader, &hdLayout);

   SetWindowPos(  hwndHeader,
                  wpos.hwndInsertAfter,
                  wpos.x,
                  wpos.y,
                  wpos.cx,
                  wpos.cy,
                  wpos.flags | SWP_SHOWWINDOW);

   ListView_EnsureVisible(hwndListView, 0, FALSE);
   }
}
```

STATUS
======

This behavior is by design.

Additional reference words: 4.00 1.30
KBCategory: kbprg kbui kbprb kbcode
KBSubcategory: UsrCtl

PRB: LoadCursor() Fails on IDC_SIZE/IDC_ICON

PSS ID Number: Q131280

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SYMPTOMS

LoadCursor() returns NULL when passed IDC_SIZE or IDC_ICON for a second parameter.

CAUSE

IDC_SIZE and IDC_ICON are obsolete. They are available only for backward compatibility. Applications marked as a version 4.0 application are not able to load these cursors under Windows 95.

STATUS

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbui kbprb
KBSubcategory: UsrRsc

PRB: LoadLibrary() Fails with _declspec(thread)

PSS ID Number: Q118816

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s versions 1.1, 1.15, 1.2, and 1.25a

SYMPTOMS
========

Your dynamic-link library (DLL) uses __declspec(thread) to allocate thread local storage (TLS). There are no problems running an application that is statically linked with the DLL's import library. However, when an application uses LoadLibrary() to load the DLL instead of using the import library, LoadLibrary() fails on Win32s with "error 87: invalid parameter". LoadLibrary() succeeds under Windows NT in this situation; however, the application cannot successfully call functions in the DLL.

CAUSE
=====

This is a limitation of LoadLibrary() and __declspec(). The global variable space for a thread is allocated at run time. The size is based on a calculation of the requirements of the application plus the requirements of all of the DLLs that are statically linked. When you use LoadLibrary(), there is no way to extend this space to allow for the thread local variables declared with __declspec(thread). This can cause a protection fault either when the DLL is dynamically loaded or code references the data.

RESOLUTION
==========

DLLs that use __declspec(thread) should not be loaded with LoadLibrary().

Use the TLS APIs, such as TlsAlloc(), in your DLL to allocate TLS if the DLL might be loaded with LoadLibrary(). If you continue to use _declspec(), warn users of the DLL that they should not load the DLL with LoadLibrary().

Additional reference words: 1.10 1.20 3.10 3.50 4.00 95
KBCategory: kbprg kbprb
KBSubcategory: BseDll

PRB: Local Reboot (CTRL+ALT+DEL) Doesn't Work Under Win32s

PSS ID Number: Q121092

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, and 1.2

SYMPTOMS

The Windows local reboot feature (CTRL+ALT+DEL) should terminate a hung application. However, if the application is a Win32-based application, using CTRL+ALT+DEL exits Windows.

CAUSE

This is a limitation of Win32s. When the VxD that handles hung applications tries to access the application stack, it uses ss:sp, instead of ss:esp, as if the application were a 16-bit application. This causes the VxD to crash, and when the VxD crashes, the whole system terminates.

STATUS

This behavior is by design.

Additional reference words: 1.10 1.15 1.20
KBCategory: kbenv kbprb
KBSubCategory: W32s

PRB: MCI Error Message with Quicktime for Windows

PSS ID Number: Q141366

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SYMPTOMS
========

When attempting to play Quicktime for Windows .mov and .qt files, MCI reports the following error:

   The specified device is not open or is not recognized by MCI

Or it returns a code of 0x107 (or 263 decimal).

NOTE: Quicktime for Windows is a third-party product manufactured by a vendor independent of Microsoft; we make no warranty, implied or otherwise, regarding its performance or reliability.

CAUSE
=====

The error indicates files required for Quicktime playback are missing from your system including the MCI Quicktime driver. Windows 95 does not automatically include support for Quicktime for Windows.

RESOLUTION
==========

Quicktime file playback requires the Quicktime for Windows files available from the Quicktime World Wide Web (WWW) home page on the Internet (http://quicktime.apple.com) or from Apple's FTP site on the Internet (ftp.apple.com). The Quicktime package includes a setup application to ease installation.

STATUS
======

Microsoft is researching this behavior and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION
================

The MCI command parser forwards commands to the specific driver for each media type (AVIVideo, Wave, CDaudio, QTWVideo, and so on). Therefore, it is up to the driver to interpret and implement different commands.

During application development, you can troubleshoot problems by simply exchanging the Quicktime file (for example, Sample.mov) with an AVI file (for example, Sample.avi) to test the sequence of MCI commands. If

the AVI file plays successfully, you have a Quicktime driver-specific issue, best handled by Apple.

Additional reference words: 4.00 avivideo mci sendstring mcisendstring
KBCategory: kb3rdparty kbmm kbprb
KBSubcategory: MMVideo

PRB: MDI Program Menu Items Changed Unexpectedly

PSS ID Number: Q74789

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

In an application for the Microsoft Windows graphical environment developed using the Windows multiple document interface (MDI), when the Window menu changes to indicate the addition of MDI child windows, another menu of the application is also changed.

For example, if the menu resembles the following before any windows are opened

```
   FILE            WINDOW
      Load            Cascade
      Save            Tile
      Save As...      Arrange Icons
      Exit
```

the menu might resemble the following after the first window is opened:

```
   FILE            WINDOW
      Load            Cascade
      1: MENU.TXT     Tile
      Save As...      Arrange Icons
      Exit            -------------
                      1: MENU.TXT
```

CAUSE

One or more menu items have menu-item identifiers that are greater than or equal to the value of the idFirstChild member of the CLIENTCREATESTRUCT data structure used to create the MDI client window.

RESOLUTION

Change the value of the idFirstChild member to be larger than any menu item identifiers.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrMdi

PRB: Memory DC Produces Monochrome Images

PSS ID Number: Q139165

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

When you create and use a memory Device Context (DC) to draw and store GDI images, the bitmap displayed when transferred from the memory device context to the physical display is monochrome. All colors are converted to either black or white.

CAUSE
=====

When creating a compatible bitmap for the memory DC, the handle to the memory DC is used as the first parameter to the call to CreateCompatibleBitmap(). This creates a monochrome bitmap because a memory DC created with CreateCompatibleDC() is given a 1x1 monochrome bitmap as its default bitmap.

For example, the following code creates a monochrome bitmap and selects it into a memory DC:

```
HDC hdc, hdcMem;
HBITMAP hBitmap;

hdc = GetDC(hWnd);
hdcMem = CreateCompatibleDC(hdc);
hBitmap = CreateCompatibleBitmap(hdcMem, 400, 400);
SelectObject(hdcMem, hBitmap);
SetTextColor(hdcMem, RGB(0, 0, 255));
```

RESOLUTION
==========

Send a physical screen DC to the CreateCompatibleBitmap() function rather than the memory DC that you plan on selecting the bitmap into.

The following example code properly sets up a color memory DC with a color bitmap with the same bit depth as the physical display:

```
HDC hdc, hdcMem;
HBITMAP hBitmap;
```

```
   hdc = GetDC(hWnd);
   hdcMem = CreateCompatibleDC(hdc);
   hBitmap = CreateCompatibleBitmap(hdc, 400, 400);
   SelectObject(hdcMem, hBitmap);
   SetTextColor(hdcMem, RGB(0, 0, 255));
```

STATUS
======

This behavior is by design.

Additional reference words: 3.10 4.00
KBCategory: kbgraphic kbprb kbcode
KBSubcategory: GdiBmp

PRB: Menus for Notification Icons Don't Work Correctly

PSS ID Number: Q135788

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When you display a context menu for a Notify Icon (see Shell_NotifyIcon), clicking anywhere besides the menu or the window that created the menu (if it is visible) doesn't cause the menu to disappear. When this behavior is corrected, the second time this menu is displayed, it displays and then immediately disappears.

RESOLUTION

To correct the first behavior, you need to make the current window the foreground window before calling TrackPopupMenu or TrackPopupMenuEx.

The second problem is caused by a problem with TrackPopupMenu. It is necessary to force a task switch to the application that called TrackPopupMenu at some time in the near future. This can be accomplished by posting a benign message to the window or thread.

The following code will take care of all of this:

```
   SetForegroundWindow(hDlg);

   // Display the menu
   TrackPopupMenu(   hSubMenu,
                     TPM_RIGHTBUTTON,
                     pt.x,
                     pt.y,
                     0,
                     hDlg,
                     NULL);

   PostMessage(hDlg, WM_USER, 0, 0);
```

STATUS

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrMisc

PRB: Messages Sent to Mailslot Are Duplicated

PSS ID Number: Q127905

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

One application creates a mailslot using CreateMailSlot() and reads from it using ReadFile(). A second application opens the mailslot using CreateFile() and writes to it using WriteFile(). The second application writes one message to the mailslot, but the first application receives three duplicates of the message.

CAUSE

This is expected behavior if you have three network transports loaded. There is no way to know which transport should be used to deliver to a given mailslot on a remote machine, so all transports are used.

RESOLUTION

Send a unique ID at the beginning of each message. The listening end can detect duplicates and delete them. If you have multiple clients sending messages, their messages may be interleaved in the mailslot. You may need to track which client sent which message last, in order to successfully detect duplicates.

STATUS

This behavior is by design.

Additional reference words: 3.50 4.00 95
KBCategory: kbprg kbprb
KBSubcategory: BseIpc

PRB: Module Relocation Fixups in Shared Sections Cause Problem

PSS ID Number: Q137235

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK)

SYMPTOMS
========

If a shared section in a 32-bit PE module contains relocation fixups, those fixups are applied to all instances of that shared section. This implies that all instances of a module with such a section must be loaded at the module's preferred image base; otherwise, the resulting relocated values will be invalid. This is true in both Windows 95 and Windows NT.

CAUSE
=====

Because an explicitly shared section shares the same physical pages with all instances of that module, fixups that are applied in a second or subsequent instance of the module overwrite the values from the first instance. The resulting values will be invalid for either one or all instances.

RESOLUTION
==========

Do not place relocatable values in a shared section. A module can be forced to be non-relocatable by removing fixup records, although this may result in load failures for that module if it can't be loaded at its preferred address.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

An example of something that causes a relocation fixup is a pointer in a shared section. Storing hInstance variables in a shared section is another.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbprb
KBSubcategory: BseMm

PRB: Most Common Cause of SetPixelFormat() Failure

PSS ID Number: Q126019

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

SetPixelFormat() will fail with incorrect class or window styles.

CAUSE
=====

Win32-based applications that use Microsoft's implementation of OpenGL to render onto a window must include WS_CLIPCHILDREN and WS_CLIPSIBLINGS window styles for that window.

RESOLUTION
==========

Include WS_CLIPCHILDREN and WS_CLIPSIBLINGS window styles when in a Win32-based application, you use Microsoft's implementation of OpenGL to render onto a window.

Additionally, the window class attribute should not include the CS_PARENTDC style. The two window styles can be added to the dwStyles parameter of CreateWindow() or CreateWindowEX() call. If MFC is used, override PreCreateWindow() to add the flags. For example:

```
    BOOL CMyView::PreCreateWindow(CREATESTRUCT& cs)
    {
        cs.style |= (WS_CLIPCHILDREN | WS_CLIPSIBLINGS);

        return CView::PreCreateWindow(cs);
    }
```

For more information, please refer to "comments" section of the online documentation on SetPixelFormat.

STATUS
======

This behavior is by design.

Additional reference words: 3.50 4.00 95
KBCategory: kbgraphic kbprb
KBSubcategory: GdiOpenGL

PRB: MoveFile Fails to Move UNC Names When NETX Is Installed

PSS ID Number: Q147437

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95

NOTE: Some products mentioned in this article are manufactured by vendors independent of Microsoft; we make no warranty, implied or otherwise, regarding these products' performance or reliability.

SYMPTOMS

When the network client is the real-mode Netware NETX redirector for Netware 3.11, MoveFile() fails to rename files and directories when the source and destination names are specified in Universal Naming Convention (UNC) format. The return value of MoveFile() is FALSE, but GetLastError() reports an error code of NO_ERROR (0).

The following code demonstrates the problem:

```
MoveFile ("\\\\testsrv\\testshare\\srcdir",
          "\\\\testsrv\\testshare\\destdir");
```

This problem does not occur with the NETX for Netware 3.12 and later or with protected-mode Netware clients.

CAUSE

NETX for Netware 3.11 does not support renaming files and directories using UNC names.

RESOLUTION

There are several possible workarounds for this problem:

- Map a drive letter to the server and share. Then use this drive letter to do the rename. Use WNetAddConnection() or WNetAddConnection2() to map the drive letter.

  -or-

- Copy the source directory (and all subdirectories) to the destination directory, and then delete the source directory. This operation will be slow if the source directory contains a lot of files and subdirectories.

  -or-

- Upgrade to the protected-mode Netware client, or use VLM instead of

NETX.

STATUS

This behavior is by design.

MORE INFORMATION

Although the bug occurs specifically when MoveFile() is used, some runtime library functions call MoveFile(), and thus exhibit the same behavior. The most notable is the C runtime library's rename() function. For information about how your particular compiler's runtime library implements rename(), refer to the documentation provided with your compiler.

Additional reference words: 4.00 move rename Novell 3.11
KBCategory: kb3rdparty kbprb
KBSubcategory: NtwkMisc

PRB: Moving or Resizing the Parent of an Open Combo Box

PSS ID Number: Q76365

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS

When the user resizes or moves the parent window of an open drop-down combo box, the list box portion of the combo box does not move.

CAUSE

The list box portion of the combo box does not receive a move message. Therefore, it remains on the screen at its original position.

RESOLUTION

Close the drop down list before the combo box is moved. To perform this task, during the processing of the WM_PAINT message, send the combo box a CB_SHOWDROPDOWN message with the wParam set to FALSE.

Additional reference words: 3.00 3.10 3.50 4.00 95 combobox
KBCategory: kbui kbprb
KBSubcategory: UsrCtl

PRB: MS-SETUP Uses \SYSTEM Rather Than \SYSTEM32

PSS ID Number: Q98888

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SYMPTOMS

Call GetWindowsSysDir() in the SETUP.MST file of a 16-bit setup application. The return value is C:\WINNT\SYSTEM\ instead of C:\WINNT\SYSTEM32\. Note that this doesn't happen with the 32-bit Setup Toolkit.

CAUSE

Windows on Win32 (WOW) returns the SYSTEM directory, not the SYSTEM32 directory, to 16-bit applications such as MS-SETUP. This is done for compatibility reasons.

RESOLUTION

Determine whether the setup code is being run under WOW or Windows version 3.1 by checking the WF_WINNT bit (0x4000) in the return from GetWinFlags(). Choose either the return from GetWindowsSysDir() or <winows dir>\system32 as appropriate.

MORE INFORMATION

Note that there are additional considerations for network installs for Win32s, because the SYSTEM directory may not be a branch off of the Windows directory.

Additional reference words: 3.10 3.50
KBCategory: kbtool kbprb
KBSubcategory: TlsMss

PRB: Named Pipe Write() Limited to 64K

PSS ID Number: Q119218

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS
========

WriteFile() returns FALSE and GetLastError() returns ERROR_MORE_DATA when WriteFile() writes to a message-mode named pipe using a buffer greater than 64K.

CAUSE
=====

There is a 64K limit on named pipe writes.

RESOLUTION
==========

The error is different from ERROR_MORE_DATA on the reader side, where bytes have already been read and the operation should be retried for the remaining message. The real error is STATUS_BUFFER_OVERFLOW. No data is transmitted; therefore, the write operation must be retried using a smaller buffer.

Additional reference words: 3.10 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseIpc

PRB: NetBIOS Command NCBSEND Gets Return Code Error 0x3C

PSS ID Number: Q123457

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

SYMPTOMS

While opening sessions, you start to get error 0x3C (lock of user area failed) in the NcbSend command call.

CAUSE

An application in Windows NT uses the NcbListen command to accept NetBIOS calls. After a call is received, the application uses the NcbSend command to send data back.

If the computer running Windows NT has 32 megabytes of main memory, an application can request a large number (for example, 128) of sessions, by using the NcbReset command, without difficulty.

However, with only 16 megabytes of main memory, an application can request only a moderate number (for example, 80) of sessions. If more sessions are opened they start to get error 0x3C (lock of user area failed) in the NcbSend command call. The error persists until some of the sessions that are currently open are closed, at which time NcbSend will get a good return status.

STATUS

This behavior is by design. The system stops the process from using so many resources that it jeopardizes the performance of other applications and/or the system itself.

Additional reference words: 3.10 3.50
KBCategory: kbprg kbenv kbprb
KBSubcategory: NtwkNetbios

PRB: Netbios RESET Cannot Be Called with Pending Commands

PSS ID Number: Q125659

The information in this article applies to:

- Microsoft Win32s, versions 1.2 and 1.25

SYMPTOMS

A Win32-based application cannot call the Netbios RESET command as long as there are asynchronous commands still pending.

CAUSE

This behavior is by design in Win32s.

RESOLUTION

The application should cancel all pending commands, and wait for the post routines of all the pending commands to be called. After that, the application can issue a Netbios RESET command.

STATUS

This behavior is by design but may be designed differently in a future version of Win32s.

Additional reference words: 1.20 1.25
KBCategory: kbprg
KBSubcategory: W32s

PRB: NetDDE Fails to Connect Under Windows 95

PSS ID Number: Q131025

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

An application using Network Dynamic Data Exchange (NetDDE) fails to connect to another DDE application on another computer.

CAUSE
=====

One of the reasons could be that NETDDE.EXE (a system component) is not running.

Network Dynamic Data Exchange (NetDDE) allows two DDE applications to communicate with each other over the network. In Windows for Workgroups, the NETDDE.EXE (a system component) was loaded by default. However under Window 95, NETDDE.EXE is not loaded by default.

RESOLUTION
==========

An application using the netDDE services should check if the netDDE system component is loaded. if NETDDE.EXE isn't running, the application should run it.

Sample Code

The following sample code checks to see if NETDDE.EXE is loaded and tries to load it if necessary. The sample code works for both 32-bit and 16-bit applications.

```
BOOL IsNetDdeActive()
{
HWND  hwndNetDDE;

// find a netDDE window
hwndNetDDE = FindWindow("NetDDEMainWdw", NULL);
// if exists then NETDDE.EXE is running
if(NULL == hwndNetDDE)
   {
   UINT  uReturn;
   // otherwise launch the NETDDE.EXE with show no active
   uReturn = WinExec("NETDDE.EXE", SW_SHOWNA);
   // if unsucessful return FALSE.
```

```
   if(uReturn <= 31)
      return FALSE;
   }
// NetDDE is running
return TRUE;
}
```

STATUS
======

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbui kbnetwork kbcode kbinterop
KBSubcategory: Usrdde

PRB: Number Causes Help Compiler Invalid Context ID Error

PSS ID Number: Q85490

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

When the Windows Help Compiler compiles a help project file, the following error message appears:

```
   Error P1083: Invalid context identification number
```

CAUSE

The representation of the context identification number begins with a zero and contains the digit 8 or 9.

RESOLUTION

Remove the leading zeros from the number.

MORE INFORMATION

The Windows Help Compiler parses a number that has a leading zero as an octal number. C compilers also interpret numbers in this manner. Only the digits 0 through 7 are legal in an octal number.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsHlp

PRB: Only One Language Can Be Used on MessageBoxEx Pushbuttons

PSS ID Number: Q152670

The information in this article applies to:

- Microsoft Win32 Software Development Kit for:
  - Microsoft Windows NT, version 3.51
  - Microsoft Windows 95, version 4.0

STMPTOMS

When MessageBoxEx is called, regardless of the value of wLanguageID (the last parameter of MessageBoxEx), the pushbutton text can only be displayed in the native language of the operating system.

On Windows NT, if the wLanguageID is the native language ID of the operating system or the wLanguageID is MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), the pushbutton text is displayed in the native language of that version of NT. Otherwise, the pushbuttons are blank.

For example, on US Windows NT 3.51:

```
MessageBoxEx( hWnd, "string", "title", MB_RETRYCANCEL,
      MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US));
```

would display a MessageBox with two buttons reading: "retry" and "cancel."

However, on the same operating system:

```
MessageBoxEx( hWnd, "string", "title", MB_RETRYCANCEL,
      MAKELANGID(LANG_FRENCH, SUBLANG_FRENCH));
```

would display a MessageBox with two blank buttons.

On Windows 95, regardless of what is specified in wLanguageID, the pushbutton text is in the native language of that version of Windows 95. For example, on US Windows 95, both:

```
MessageBoxEx( hWnd, "string", "title", MB_RETRYCANCEL,
      MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US));
```

-and-

```
MessageBoxEx( hWnd, "string", "title", MB_RETRYCANCEL,
      MAKELANGID(LANG_FRENCH, SUBLANG_FRENCH));
```

would display a Message Box with two buttons reading: "retry" and "cancel."

STATUS

This behavior is by design.

Additional reference words: 3.51 4.00 resource NLS
KBCategory: kbprg
KBSubcategory: wintldev

PRB: OpenFile API returns ERROR_BUFFER_OVERFLOW Error

PSS ID Number: Q137233

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

If the OpenFile API is passed a path that is longer than MAXPATHNAME, OpenFile returns ERROR_BUFFER_OVERFLOW.

One of the parameters to OpenFile is a pointer to the pathname to open. Another parameter is a pointer to an OFSTRUCT structure that will receive information. OFSTRUCT contains a fixed size buffer, szPathName that holds MAXPATHNAME bytes. A path longer than that is not valid.

RESOLUTION

Consider using CreateFile instead of OpenFile.

STATUS

This behavior is by design.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbprb
KBSubcategory: BseFileio

PRB: OpenFile Does Not Work Well with SetFileAPIsToOEM

PSS ID Number: Q137201

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

SYMPTOMS

After you call SetFileApisToOEM, OpenFile is supposed to accept OEM strings for file manipulation. However, giving OpenFile an extended character produces an unexpected filename. For example, with char 190, the filename contains a '+' instead of the expected character.

CAUSE

OpenFile converts the given OEM string internally to ANSI before manipulating the filename or passing it down to the filesystem. This conversion is not always successful, as not all OEM chars have equivalents in the ANSI set.

RESOLUTION

Use Createfile to open the file. CreateFile does not convert OEM to ANSI, so this problem does not occur.

STATUS

This behavior is by design.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbprb
KBSubcategory: BseFileio

PRB: Oracle7 for Win32s Hangs When Initialize Database Manager

PSS ID Number: Q127760

The information in this article applies to:

- Microsoft Win32s versions 1.00, 1.10, 1.15, 1.20, and 1.25a

---

SYMPTOMS

There is a known problem with using Oracle7 for Win32s with the Windows Sound System version 2.0. The problem occurs when starting up a database in the Database Manager. Oracle7 may hang during this process. The machine will need to be rebooted.

CAUSE

The SNDEVNTS.DRV file is causing the problem by performing stack checking on the application stacks of Oracle7. The stack checking is corrupting the application stack of Oracle7 causing the application to hang.

RESOLUTION

Download SEVNT022.EXE, a self extracting file from the Microsoft Software Library (MSL) available on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download SEVNT022.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get SEVNT022.EXE

Follow the installation directions contained in the README.TXT file.

Additional reference words: 1.00 1.10 1.20 SNDEVNTS.DRV
KBCategory: kb3rdparty kbfile kbprb
KBSubcategory: W32s

PRB: Page Fault in WIN32S16.DLL Under Win32s

PSS ID Number: Q115082

The information in this article applies to:

- Microsoft Win32s, versions 1.1 and 1.15

SYMPTOMS

Start two instances of the Win32 application under Win32s. Close one instance, then close the other instance. A page fault is generated in WIN32S16.DLL. Under Win32s, version 1.1, a dialog box appears. Under Win32s, version 1.15, the error only appears in the output from the debug version of Win32s.

CAUSE

This problem can be caused by using the static C run-time (CRT) libraries or using MSVCRT10.DLL. Note that even if the application does not make any CRT calls, one of its DLLs may call the CRT initialization and cleanup code. The CRT code makes the following sequence of API calls:

```
DeleteCriticalSection
DeleteCriticalSection
DeleteCriticalSection
TlsFree
VirtualFree
VirtualFree
VirtualFree
```

When the second application instance terminates, it faults before it makes the call to TlsFree(). The CRT has two blocks, one that contains strings from the environment and one that contains pointers to the first blocks. These are allocated by the first process that attach to the DLL. Other processes that attach do not allocate these blocks. When processes are terminated, these two blocks are freed. However, this succeeds only when the process that owns the memory frees them. Any other process that tries to access these blocks will fail.

RESOLUTION

Because there is no instance data by default under Win32s, DLLs should use the CRT in a DLL instead of linking to the CRT statically. MSVCRT10.DLL (which comes with Visual C++) is not compatible with Win32s because MSVCRT10.DLL falsely assumes that Win32s implements instance data, which is only available on Windows NT. Therefore, until an updated MSVCRT.LIB file is released, use CRTDLL.LIB (which comes with the Win32 SDK) because Win32s has its own CRTDLL.DLL file that was specifically designed for this use.

Microsoft Visual C++ 2.0 contains two versions of MSVCRT20.DLL: one version

is intended for use on Windows NT, the other is intended for use on Win32s. To avoid this problem, use and ship the Win32s version of MSVCRT20.DLL in your application.

MORE INFORMATION

The real problem is that memory is allocated for several applications. The allocation is done by the first application. When this application terminates, it takes with it all of its memory. This means that each time the remaining applications try to access this memory, an error occurs. Symptoms include data corruption, hanging, or the WIN32S16.DLL page fault mentioned above.

Additional reference words: 1.10 1.15
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: PaintRgn() Fills Incorrectly with Hatched Brushes

PSS ID Number: Q82169

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SYMPTOMS

When the TRANSPARENT background mode and a mapping mode other than MM_TEXT are selected and an application calls the PaintRgn() API to fill a complex region with a hatched brush, a disconnected pattern results.

CAUSE

The Windows Graphics Device Interface (GDI) draws a complex region by filling the individual rectangles that make up the region. The code to compute the position of each rectangle on the screen fails when the screen coordinates are not in units of pixels. The error is visible when a hatched brush style is used in TRANSPARENT mode.

RESOLUTION

When a hatched brush and TRANSPARENT background mode are required, use the MM_TEXT mapping mode.

Additional reference words: 3.10 4.00 95
KBCategory: kbgraphic kbprb
KBSubcategory: GdiDrw

PRB: Poor TCP/IP Performance When Doing Small Sends

PSS ID Number: Q126716

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

When doing multiple sends of less than the Maximum Transmission Unit (MTU), you may see poor performance. On an Ethernet network, the MTU for TCP/IP is 1460 bytes.

CAUSE

When an application does two sends of less than a transport MTU, the second send is delayed until an ACK is received from the remote host. The delay occurs in case the application does another small send. TCP can then coalesce the two small sends into one larger packet. This concept of collecting small sends into larger packets is called Nagling.

RESOLUTION

There are a number of ways to avoid Nagling in an application. Here are two. The second is more complex but gives a better performance benefit:

- Set the TCP_NODELAY socket option for the socket. This tells TCP/IP to send always, regardless of packet size. This will result in sub-optimal use of the physical network, but it will avoid the delay of waiting for an ACK.

- Send larger blocks of data. The send() API call, when you include the overhead of the other network components involved, costs a couple of thousand instructions. One large send() call will be more efficient than two smaller send() calls, even if you need to do some buffer copies.

  Sending larger data blocks will also result in more efficient use of the physical network because packets will typically be larger and less numerous. This option is much better than the first (enabling TCP_NODELAY) and should be used if at all possible.

On Windows NT 3.51, if you are sending files, you should use the new TransmitFile() API. This call reads the file data directly from the file system cache and sends it out over the wire. The TransmitFile() call can also take a data block that will be sent ahead of the file, if desired.

REFERENCES

More information about Nagling and the Nagle algorithm can be found in RFC 1122.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork kbnetwork kbprb
KBSubcategory: NtwkWinsock

PRB: Pressing the ENTER Key in an MDI Application

PSS ID Number: Q99799

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS

In a standard Microsoft Windows version 3.1 multiple document interface (MDI) application, when a minimized MDI child is active and the user presses the ENTER key, the child is not restored.

This is inconsistent with other MDI applications, such as File Manager and Program Manager. An MDI child in one of these applications is restored when it is the active MDI child and the ENTER key is pressed. When a normal Windows-based application is minimized and the user presses ENTER, that application is restored to a normal state.

RESOLUTION

One quick workaround to this problem is to create an accelerator for the ENTER key and restore the minimized MDI child when the key is pressed.

MORE INFORMATION

It may be desirable to implement the same restore feature that File Manager and Program Manager have implemented in order to enable the user to restore an MDI child by pressing the ENTER key. If this feature is implemented, then the MDI application can be consistent with other popular applications such as File Manager, Microsoft Excel, and Microsoft Word.

To achieve this effect in an MDI application, create an accelerator in the accelerator table of the resource file for the application. This can be done as follows:

```
MdiAccelTable ACCELERATORS
     {
     .  .  .
     .  .  .
     VK_RETURN, IDM_RESTORE, VIRTKEY
     }
```

After this accelerator has been installed in the MDI application, each time the ENTER key is pressed by the user, an IDM_RESTORE command will be sent to the MDI frame window's window procedure through a WM_COMMAND message. When the MDI frame receieves this message, its window procedure should retrieve a handle to the active MDI child and determine if it is minimized. If it is minimized, then it can restore the MDI child by sending the MDI client a WM_MDIRESTORE message. This can all be done with the following code:

```
        case IDM_RESTORE:
           {
           HWND hwndActive;

           hwndActive = SendMessage(hwndClient,WM_MDIGETACTIVE,0,0L);
           if (IsIconic(hwndActive))
              SendMessage(hwndClient,WM_MDIRESTORE,hwndActive,0L);
           break;
           }
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrMdi

PRB: Print Common Dialog Box Doesn't Display Help Button

PSS ID Number: Q140271

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SYMPTOMS

When an application invokes the print common dialog box and specifies the PD_SHOWHELP flag, the help button is not displayed on the dialog box when the application is run in Windows 95.

CAUSE

The Windows 95 common dialog code does not create the help button even though the PD_SHOWHELP is supplied.

RESOLUTION

To overcome this behavior, an application needs to install a common dialog hook function and create the help button itself. The Help button event handling code in Windows 95 will function properly. The following code fragment demonstrates how to create and position the Help button on the Print common dialog.

This behavior also occurs with the Page Setup common dialog box. The following code will work on this dialog box also, but the positioning of the button will have to be adjusted to place it properly.

```
/*********************************************************************
   DoPrintDialog()
*********************************************************************/

BOOL DoPrintDialog(HWND hwndParent, BOOL bSetup)
{
PRINTDLG pd;
BOOL     bReturn;

/*
initialize PRINTDLG structure
*/
ZeroMemory(&pd, sizeof(pd));

pd.lStructSize = sizeof(PRINTDLG);
pd.hwndOwner = hwndParent;
pd.hDevMode = NULL;
pd.hDevNames = NULL;
pd.nFromPage = 0;
pd.nToPage = 0;
```

```
pd.nMinPage = 0;
pd.nMaxPage = 0;
pd.nCopies = 0;
pd.hInstance = g_hInst;

if(bSetup)
   {
   /*
   display the print setup dialog box
   */
   pd.Flags =  PD_RETURNDC |
               PD_SHOWHELP |
               PD_PRINTSETUP |
               PD_ENABLESETUPHOOK;

   pd.lpfnSetupHook = PrintDlgHookProc;
   pd.lpSetupTemplateName = NULL;
   }
else
   {
   /*
   display the regular print dialog box
   */
   pd.Flags =  PD_RETURNDC |
               PD_SHOWHELP |
               PD_ENABLEPRINTHOOK;

   pd.lpfnPrintHook = PrintDlgHookProc;
   pd.lpPrintTemplateName = NULL;
   }

bReturn = PrintDlg(&pd);

if(bReturn)
   {
   }

return bReturn;
}

/*********************************************************************
   PrintDlgHookProc()
*********************************************************************/

BOOL CALLBACK PrintDlgHookProc(  HWND hWnd,
                                 UINT uMessage,
                                 WPARAM wParam,
                                 LPARAM lParam)
{
switch (uMessage)
   {
   case WM_INITDIALOG:
      {
      /*
      lParam is a pointer to the PRINTDLG structure
```

```
*/
LPPRINTDLG  lppd = (LPPRINTDLG)lParam;

/*
Only create the help button if it has been specified and it
doesn't already exist - pshHelp is defined in Dlgs.h.
*/
if(   (lppd->Flags & PD_SHOWHELP) &&
      (NULL == GetDlgItem(hWnd, pshHelp)))
   {
   RECT  rc,
         rcGroup;
   HWND  hwndHelp;
   HFONT hFont;

   /*
   Get the rectangle of the OK button as a template
   */
   GetWindowRect(GetDlgItem(hWnd, IDOK), &rc);
   MapWindowPoints(HWND_DESKTOP, hWnd, (LPPOINT)&rc, 2);

   /*
   Get the rectangle of the group box
   */
   GetWindowRect(GetDlgItem(hWnd, grp4), &rcGroup);
   MapWindowPoints(HWND_DESKTOP, hWnd, (LPPOINT)&rcGroup, 2);

   /*
   Put the width in right and the height in left
   */
   rc.right = rc.right - rc.left;
   rc.bottom = rc.bottom - rc.top;

   /*
   Align the button - this is reliable for the print and setup
   dialogs
   */
   rc.left = rcGroup.left;

   /*
   Create the help button
   */
   hwndHelp = CreateWindow(
      "button",
      "&Help Button",
      WS_CHILD | WS_VISIBLE | WS_TABSTOP | BS_PUSHBUTTON,
      rc.left,
      rc.top,
      rc.right,
      rc.bottom,
      hWnd,
      (HMENU)pshHelp,
      (HINSTANCE)GetWindowLong(hWnd, GWL_HINSTANCE),
      NULL);
```

```
         /*
         Get the font from the OK button and set the font of the help
         button
         */
         hFont = (HFONT)SendDlgItemMessage(hWnd, IDOK, WM_GETFONT, 0, 0);
         SendMessage(hwndHelp, WM_SETFONT, (WPARAM)hFont, 0);
         }
      }
      return TRUE;

   }
return FALSE;
}
```

STATUS
======

This behavior is by design.

Additional reference words: 1.30 4.00
KBCategory: kbui kbprb kbcode kbhowto
KBSubcategory: UsrCtl W32s

PRB: Printer Font too Small with ChooseFont() Common Dialog

PSS ID Number: Q89544

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

In an application for the Microsoft Windows graphical environment, the user selects a printer font through the ChooseFont() common dialog box function. When the application calls the CreateFontIndirect() to create the selected font, even though the style and face name are correct, the point size is much too small.

CAUSE

The LOGFONT structure returned to the application is based on screen metrics--even when the user selects a printer font. Because the resolution (dots per inch) on a printer is generally much greater than that of the screen, the resulting printer font is smaller than desired.

RESOLUTION

Modify the lfHeight member of the LOGFONT data structure according to the printer metrics.

MORE INFORMATION

The following code demonstrates how to modify the lfHeight member. The lpcf variable contains a pointer to the CHOOSEFONT data structure. The hDC member of the CHOOSEFONT data structure is a handle to the printer device context.

```
   if (ChooseFont(lpcf))
   {
      if (lpcf->nFontType & PRINTER_FONT)
      {
         iLogPixelsy = GetDeviceCaps(lpcf->hDC, LOGPIXELSY);
         lpcf->lpLogFont->lfHeight =
             MulDiv(-iLogPixelsy, (lpcf->iPointSize / 10), 72);
         hPrinterFont =
             CreateFontIndirect((LPLOGFONT)(lpcf->lpLogFont));
      }
```

```
      else
      {
         // Create screen font
      }
   }
   else
   {
      // Process common dialog box error
   }
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrCmnDlg

PRB: Private Button Class Can't Get BM_SETSTYLE in Windows 95

PSS ID Number: Q130951

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When an application creates a new button class, the new buttons do not receive BM_SETSTYLE messages under Windows 95.

CAUSE

In previous versions of Windows, the new button had only to return DLGC_BUTTON in response to the WM_GETDLGCODE message. This was all that was required to identify the window as a "button."

However, in Windows 95, returning DLGC_BUTTON to WM_GETDLGCODE is no longer sufficient to identify the window as a "button." The dialog manager code in Windows 95 is implemented in 16 bits. When a message is dispatched to a 32-bit window, the system automatically generates a thunk. Because the system does not know that the new class is actually a "button," it does not automatically perform the thunk - so the BM_SETSTYLE messages are not sent.

RESOLUTION

To tell the system to treat the window as a "button," the window must call one of the following APIs at least once:

IsDlgButtonChecked
CheckRadioButton
CheckDlgButton

The preferable method for doing this is to call IsDlgButtonChecked during the WM_CREATE message. Once this is done, the window will receive all standard button messages.

STATUS

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbui kbprb
KBSubcategory: UsrCtl

PRB: Problem with Shared Data Sections in DLLs.

PSS ID Number: Q147136

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:

    - Windows NT versions 3.5 and 3.51
    - Windows 95 version 4.0

SYMPTOMS
========

Because Win32 uses path and file names to identify loaded modules, running an application that uses a DLL with a shared data section may not work correctly if the DLL will be loaded using different paths.

RESOLUTION
==========

A memory-mapped file should be used to share data between processes.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

In Win32, module names include the path and file name, unlike 16-bit Windows where all modules are identified with up to eight character names.

When you use a DLL with a shared data section, the path\file name convention of identifying module names can cause shared sections to not be shared. When the DLL is loaded a second time using a different path, the operating system treats each instance of the DLL as a separate DLL. Because the DLL is loaded twice as separate DLLs, no sections are shared, not even code sections.

For example, when you map network drives E: and F: to the same share and from each drive run an instance of an application that shares data by using a shared data section in a DLL, the DLL is loaded separately with module names like E:\path\filename and F:\path\filename. Each DLL will have its own shared data section and therefore the processes will each see a different copy of the data. In fact, shared data sections are implemented as memory mapped files, so in this case, each instance of the DLL sets up its own memory mapped file for a shared data section.

This problem can be diagnosed by putting a call to GetModuleFileName in the DLL and examining the library names of each instance of the DLL. If the names are different, shared sections will not work for the processes calling them. The reason a file mapping is immune to the different path

names is because each process accesses a file mapping by name. File mapping names are constant and independent of file paths.

Additional reference words: 3.50 4.00
KBCategory: kbprg kbprb
KBSubcategory:

PRB: Problems with the Microsoft Setup Toolkit

PSS ID Number: Q106382

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SYMPTOMS
========

1. When the /zi option is used with the Win32 DSKLAYT2.EXE to provide compression, it causes an access violation.

2. The Win32 Setup Toolkit does not contain a setup bootstrapper to copy the needed setup files to a temp drive and run the Setup program. Setup runs from floppy disks.

3. Install programs for applications that may run on Win32s must be created with the 16-bit version of the Setup Toolkit if the installation program will install Win32s. However, the 16-bit DSKLAY2.EXE cannot read the version information in a Win32 binary.

4. The Win32 DSKLAYT.EXE only shows 8.3 names in the list box.

5. The Setup program reports "out of memory" during installation, but there seems to be plenty of memory.

6. Installation fails from a CD-ROM. If the same files are copied from the CD to the hard disk, installation succeeds.

7. Setup programs created with the 32-bit setup toolkit will not run under Win32s.

RESOLUTIONS
===========

1. The fix for this problem is available in the Alpha SDK Update and later.

   Note that COMPRESS.EXE has been updated to use a better compression algorithm, and therefore /zi is no longer recommended for best compression. The option has been kept for compatibility reasons.

2. The bootstrapper is not necessary in a 32-bit environment. It is required for Windows because it is not possible to remove the floppy disk of a currently running Win16 application (the resources could not all be preloaded and locked).

3. If a Win32s installation is provided on a separate disk, the install program can be developed with the Win32 Setup Toolkit.

4. The program is actually a 16-bit program, and therefore it can display only the 8.3 name. Use 8.3 names for the source names and

specify that the files be renamed (using the long names) when they are installed.

5. This error can be caused when a DLL on disk 1 is needed when when a different disk is currently inserted. To work around this problem, use LoadLibrary() to load the DLL.

6. This problem was corrected in the Win32 3.5 SDK.

7. This problem was corrected in the Win32 3.5 SDK.

Additional reference words: 3.10 3.50
KBCategory: kbtool kbprb
KBSubcategory: TlsMss

PRB: Processing the WM_QUERYOPEN Message in an MDI Application

PSS ID Number: Q99411

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When a multiple document interface (MDI) child window processes the WM_QUERYOPEN message to prevent the child from being restored out of a minimized state, the system menu and restore button are not removed from the menu bar when a maximized MDI child loses the focus or is closed.

This problem occurs only when the maximized MDI child loses the focus or is closed and the focus is given to an MDI child that returns 0 (zero) on the WM_QUERYOPEN message.

CAUSE

When an MDI child is maximized, the system menu and restore button are added to the frame menu as bitmap menu items. When a maximized MDI child is destroyed or another MDI child is given the focus, the MDI child given the focus afterwards is maximized to replace the old MDI child. Windows cannot maximize an MDI child when it is processing the WM_QUERYOPEN message, and therefore the child is not maximized. Unfortunately, the system menu and restore button bitmaps are not removed from the menu bar.

RESOLUTION

To prevent this problem, restore the maximized MDI child before giving the focus to another child.

MORE INFORMATION

It may sometimes be desirable to prevent an MDI child from being restored during part or all of its life. This can be done by trapping the WM_QUERYOPEN message by placing the following code in the window procedure of the MDI child:

```
   case WM_QUERYOPEN:
      return 0;
```

Unfortunately, this causes the added restore and system menu bitmaps to remain on the menu bar when a maximized MDI child loses the focus or is closed and the focus is given to a child processing this message. The following code can be used to restore a maximized MDI child when it loses the focus:

```
     case WM_MDIACTIVATE:
        if ((wParam == FALSE) && (IsZoomed(hwnd)))
           SendMessage(hwndMDIClient, WM_MDIRESTORE, hwnd, 0L);

        return DefMDIChildProc (hwnd, msg, wParam, lParam);
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrMdi

PRB: Property Sheet w/ Multiline Edit Control Ignores ESC Key

PSS ID Number: Q130765

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS

Pressing the ESC key when the focus is on a Multiline Edit control that is a child of a property sheet page, does not dismiss the property sheet control as expected.

CAUSE

When the ESC key is pressed while focus is on a Mutiline Edit control, the IDCANCEL notification (sent with a WM_COMMAND message) is sent to the property sheet dialog proc whose template contains the Multiline Edit control. Property sheet dialog procs do not process this message, so it is not forwarded to the property sheet control.

RESOLUTION

Trap the IDCANCEL notification that is sent along with the WM_COMMAND message in the property sheet dialog proc that contains the multiline edit control. Then forward the message to the property sheet control. (The property sheet control is the parent of all the property sheet page dialogs.) The following code shows how to do this:

Code Sample

```
//
//  FUNCTION: SheetDialogProc(HWND, UINT, WPARAM, LPARAM)
//
//  PURPOSE:  Processes messages for a page in the PPT sheet control.
//
//  PARAMETERS:
//    hdlg - window handle of the property sheet
//    wMessage - type of message
//    wparam - message-specific information
//    lparam - message-specific information
//
//  RETURN VALUE:
//    TRUE -  message handled
//    FALSE - message not handled
//
```

```
LRESULT CALLBACK SheetDialogProc(HWND hdlg,
                      UINT uMessage,
                     WPARAM wparam,
                     LPARAM lparam)
{
    LPNMHDR lpnmhdr;
    HWND    hwndPropSheet;
    switch (uMessage)
    {
   case WM_INITDIALOG:

      // Do whatever initializations you have here.
      return TRUE;


   case WM_NOTIFY:

      // more code here ...

      break;


    case WM_COMMAND:

       switch(LOWORD(wparam))
      {

         case IDCANCEL:

         // Forward this message to the parent window
         // so that the PPT sheet is dismissed
            SendMessage(GetParent(hdlg),
                        uMessage,
                        wparam,
                        lparam);
         beak;

         default:
         break;
      }
      break;

   }
```

NOTE: This solution also works with wizard controls. This behavior is seen under both Windows NT and Windows 95; the solution in this article works for both platforms.

Additional reference words: 4.00 user styles
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrCtl

PRB: Quotation Marks Missing from Compiled Help File

PSS ID Number: Q110540

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

After upgrading from Word for Windows 2.x to Word for Windows 6.x and rebuilding a Windows Help file with HCP.EXE or HC31.EXE, all of the quotation marks are missing from the Help file.

CAUSE

Word 6.x uses a special .RTF keyword to represent the quotation mark character, and the Help compiler does not understand the new .RTF keyword so it drops the character. The same problem occurs for the curly double quotation mark, single quotation mark, en dash (char 150), em dash (char 151), and the bullet character (char 149).

RESOLUTION

You can prevent this problem by turning off the "Smart Quotes" option in Word for Windows. The following three steps accomplish this:

1. Choose Options from Tools menu.
2. Select the AutoFormat tab.
3. In the Replace group box, clear the "Straight Quotes with Smart Quotes" check box, and choose OK.

If you wish to include smark quotes, bullets, em-dashes, and en-dashes in a Help file, you can open the file as text only and replace the RTF keywords with their ANSI hexidecimal equivalents, which are recognized by the help compiler.

Find String	Replace String
\emdash	\'97"
\endash	\'96"
\bullet	\'95"
\rdblquote	\'94"
\ldblquote	\'93"
\rquote	\'92"
\lquote	\'91"

Make sure that there is a blank character included at the end of the Find String, but not in the Replace String.

This replacement must be made every time you edit and convert the text from document to RTF format.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsHlp

PRB: RasDial() Takes Longer to Dial Out

PSS ID Number: Q140018

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.51

SYMPTOMS
========

RasDial() takes more time to dial and authenticate a RAS (remote access server) user if the RAS server being dialed is using older RAS protocols and if the application is not using a phone book entry while calling RasDial().

CAUSE
=====

Windows NT versions 3.5 and 3.51 use PPP (point-to-point protocol) as the default protocol for dialing out. However, older versions of RAS servers namely, RAS servers for Windows NT version 3.1 and point-to-point servers for Windows for Workgroups do not implement the PPP protocol.

When an application calls RasDial() without specifying a phonebook entry, RasDial() uses the default phonebook entry settings. The default framing protocol for Windows NT versions 3.5 and 3.51 is PPP, so that is tried first. If PPP doesn't work, the old RAS framing protocol is tried. The failed attempt to connect with PPP accounts for the delay.

When a user creates a phonebook entry and dials an older RAS server, RAS on the Windows NT client tries PPP as the framing protocol, which eventually times out, and RAS falls back to old RAS protocol. RasDial() marks the phonebook entry so that the next time it is dialed old RAS protocol is tried first with fallback to PPP. Thus, the user sees the same delay on the first call, but all subsequent calls are quick.

RESOLUTION
==========

Application programmers can create an entry with RASPHONE and override the phone number at dial time, thus using the entry as a template with options appropriate for the down-level server. If it is known that the server is an older RAS server and will not be upgraded, you can clear all three check boxes for all three protocols. Using RASPHONE, on the Edit menu, click Network, and then clear the check boxes in the PPP group. This trick forces old RAS protocol and eliminates the training call.

STATUS
======

This behavior is by design.

Additional reference words: 3.51

KBCategory: kbprg kbnetwork kbprb
KBSubcategory: NtwkRAS

PRB: RegCreateKeyEx() Gives Error 161 Under Windows NT 3.5

PSS ID Number: Q117261

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

SYMPTOMS

A call to RegCreateKeyEx() is successful under Windows NT version 3.1, but the call fails with error 161 (ERROR_BAD_PATHNAME) under Windows NT version 3.5.

The sample code below demonstrates this problem.

CAUSE

This is by design. Windows NT version 3.1 allows the subkey to begin with a backslash ("\"), however Windows NT version 3.5 does not. The subkey is given as the second parameter to RegCreateKeyEx().

RESOLUTION

Remove the backslash from the beginning of the subkey name.

MORE INFORMATION

In the sample code below, RegCreateKeyEx() fails with error 161 while the string defined by SUBKEY_FORMAT_STRING begins with a backslash, but succeeds if the initial backslash is removed.

Sample Code

```
#include <windows.h>
#include <stdio.h>

#define SUBKEY_FORMAT_STRING \
"\\SYSTEM\\CurrentControlSet\\Services\\EventLog\\Application\\%s"

void main(int argc, char *argv[])
{
   DWORD dwErrorCode;
   char lpszSubKey[MAX_PATH];
   HKEY hKey;
   DWORD dwDisposition;
```

```
      sprintf( lpszSubKey, SUBKEY_FORMAT_STRING, argv[1] );

      printf( "Trying to open: %s\n", lpszSubKey );

      dwErrorCode = RegCreateKeyEx(HKEY_LOCAL_MACHINE,
                                   lpszSubKey,
                                   0,
                                   "",
                                   REG_OPTION_NON_VOLATILE,
                                   KEY_ALL_ACCESS,
                                   NULL, //Security
                                   &hKey,
                                   &dwDisposition );

      if (dwErrorCode != ERROR_SUCCESS)
         printf( "Code = %d.\n", dwErrorCode );

      RegCloseKey(hKey);
   }
```

NOTE: Double backslashes ("\\") are required in strings in C code to represent a single backslash, since a backslash ordinarily indicates the beginning of an escape sequence.

Additional reference words: 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseMisc

PRB: RegisterClass()/ClassEx() Fails If cbWndExtra > 40 Bytes

PSS ID Number: Q131288

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

Under Windows 95, a call to RegisterClass() or RegisterClassEx() returns NULL if a value greater than 40 is specified for the cbWndExtra or cbClsExtra members of the WNDCLASS or WNDCLASSEX structure.

CAUSE
=====

Windows 95 checks to see if cbWndExtra or cbClsExtra is greater than 40. If so, it outputs these debug messages and returns NULL to indicate failure:

```
   RegisterClassEx: Unusually large cbClsExtra (>40)
   RegisterClassEx: Unusually large cbWndExtra (>40)
```

RESOLUTION
==========

If more than 40 bytes are needed to store window-specific or class-specific information, an application should allocate memory. Then set the cbWndExtra or cbClsExtra to 4 bytes, and pass the pointer to the allocated memory by using SetClassLong() as follows:

```
   SetClassLong (hWnd, GCL_CBCLSEXTRA, lpMemoryAllocated);
   SetClassLong (hWnd, GCL_CBWNDEXTRA, lpMemoryAllocated);
```

The pointer can then be retrieved when needed by using GetClassLong() as follows:

```
   lpMemoryAllocated = GetClassLong (hWnd, GCL_CBCLSEXTRA);
   lpMemoryAllocated = GetClassLong (hWnd, GCL_CBWNDEXTRA);
```

STATUS
======

This behavior is by design. However, as of version 3.51, Windows NT does not have this 40-byte limitation.

Additional reference words: 4.00
KBCategory: kbui kbprb
KBSubcategory: UsrCls

PRB: RestoreDC() Fails Across Printer Pages

PSS ID Number: Q139005

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API)
  included with Microsoft Windows 95 version 4.0

SYMPTOMS
========

The RestoreDC() function returns failure when attempting to restore a printer Device Context (DC) that was saved by calling the SaveDC() function in Windows 95 version 4.0.

NOTE: This occurs only for applications marked as 4.0 applications.

CAUSE
=====

RestoreDC() fails if the specified state instance does not exist on the GDI instance stack. Specifically, RestoreDC() fails on a printer DC if the StartPage() function is called between a call to the SaveDC() and a call to RestoreDC(). The RestoreDC() function fails because the StartPage() function in Windows 95 deletes any DC state instances that have been saved for that DC.

RESOLUTION
==========

Applications use the StartPage() and EndPage() functions to indicate page separations on printer DCs. It is therefore necessary for applications written for Windows 95 to reinitialize a printer DC by selecting GDI objects and resetting mapping modes and other DC and GDI object attributes after every call to StartPage().

STATUS
======

This behavior is by design.

MORE INFORMATION
================

When Windows 95-based applications print, they spool to Enhanced Metafile (EMF) spool files by default. Spooling to an EMF requires a clean DC so that each call to the StartPage() function restores the printer DC to its default state.

In addition to restoring the default state of the DC, Windows 95 also empties the stack of DC states that may have been saved by calling the SaveDC() function. This prevents applications from setting up a DC prior to a call to StartPage() and restoring the selected objects and attributes

after the call to StartPage() with RestoreDC(). However, as long as the SaveDC() and RestoreDC() functions are called between StartPage()/EndPage() pairs, they will succeed and properly save and restore DC states for the printer DC.

Typically, applications should allocate GDI resources prior to the start of a document or page and reuse these resources to initialize a printer DC at the start of every page in a document.

Additional reference words: 4.00 Windows 95 difference blank page output
KBCategory: kbprint kbgraphic kbprb
KBSubcategory: GdiDc GdiPrn

PRB: Result of localtime() Differs on Win32s and Windows NT

PSS ID Number: Q117893

The information in this article applies to:

- Microsoft Win32s versions 1.1, 1.15, and 1.2

SYMPTOMS
========

Under Windows NT, localtime(time()) returns the correct local time. However, under Win32s, the local time returned is not correct if the TZ environment variable is set. For example, suppose that you are in the Pacific time zone (GMT-08:00) and have set tz=pst8pdt. The time returned under Win32s is off by 8 hours.

CAUSE
=====

This is by design.

The localtime() function depends on time zone information, which is not available in Win32s. This is the reason that the Win32 API GetLocalTime() is not supported under Win32s. The C Run-time functions, like localtime(), use the tz environment variable for time zone information.

The time() function returns the current local time under Win32s, then the call to localtime() adjusts the time by the offset of your time zone from GMT, which it finds by reading the tz environment variable.

Under Windows NT, time() and GetSystemTime() return GMT, therefore localtime(time()) is the current local time.

RESOLUTION
==========

To get the current local time under both Win32s and Windows NT, use the following code to clear the tz environment variable and get the time:

```
_putenv( "TZ=" );
_tzset();

localtime( time() );
```

Note that _putenv() affects only the tz environment variable for the application. All other applications use the global environment settings and make their own modifications.

Additional reference words: 1.10 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: RichEdit Control Doesn't Save REO_* Flags

PSS ID Number: Q135987

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS

A program can embed an OLE object in a RichEdit control by using one of the REO_* flags (for example, REO_BELOWBASELINE) to change the standard attributes for the embedded object. However, when the file is saved by using the EM_STREAMOUT message and read back into the control by using the EM_STREAMIN message, the REO_* flags are not applied to the object.

CAUSE

The RichEdit control does not save the REO_* flags when it writes the object to a file.

WORKAROUND

To work around this behavior, have your program handle saving the OLE objects contained in the RichEdit control to disk manually. Then the program can have the RichEdit control save the rest of the contents without the OLE objects by sending the EM_STREAMOUT message to the RichEdit control with the SF_RTFNOOBJS flag. By specifying the SF_RTFNOOBJS flag, the RichEdit control will leave a space character where the object was, so it is important that your program save the position of the OLE objects as well as the actual objects.

STATUS

This behavior is by design.

Additional reference words: 4.00 1.30 RTF Edit Windows 95
KBCategory: kbui
KBSubcategory: UsrCtl

PRB: RichEdit Control Hides Mouse Pointer (Cursor)

PSS ID Number: Q131381

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SYMPTOMS

When the mouse pointer (cursor) is over a RichEdit control and the user starts typing in the control, the RichEdit control hides the pointer.

CAUSE

When the user is typing in a RichEdit control, the control automatically hides the mouse pointer if it is is over the control. The control does this intentionally so the cursor does not obscure the text in the control. When the mouse is moved again, the pointer re-appears.

STATUS

This behavior is by design.

Additional reference words: 4.00 95 1.30
KBCategory: kbui kbprb
KBSubcategory: UsrCtl

PRB: RoundRect() and Ellipse() Don't Match Same Shaped Regions

PSS ID Number: Q119455

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When CreateRoundRectrRgn() is used to create a region with the shape of a rectangle that has rounded corners and RoundRect() is called with the same parameters to draw the same rectangle that has rounded corners, the calculated region does not match the drawn rectangle. The same can be said of the ellipses created by CreateEllipticRgn() and Ellipse().

CAUSE

This behavior is because of the design on Windows. The mismatch between fills and frames is because of the way that the boundaries and fills must be specified in order to get polygons to fit together properly. Windows NT duplicates this behavior for compatibility.

RESOLUTION

Perform the fill first, then draw the frame. Some of the frame pixels will overwrite fill pixels and some will not; however, there will be no gap between the frame and the fill, and the fill will not extend past the frame. Use CreateRoundRectRgn() or CreateEllipticRgn() for the fill and RoundRect() or Ellipse, respectively, for the frame. Use the same parameters for both the region API and the filled-shape API.

NOTE: If you use a NULL pen when drawing the filled shape, the pixels will match those drawn by creating a region through the corresponding region API and then calling FillRgn() with the same parameters. It draws the frame with the pen from the filled-shape API that causes the discrepancy.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic kbprb
KBSubcategory: GdiDrw

PRB: RPC App Using Auto Handles Can't Call Remote Procedure

PSS ID Number: Q140015

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) version 3.51

SYMPTOMS
========

An RPC application using auto handles cannot call a remote procedure on a server that is in a different Windows NT computer domain than the client workstation's Windows NT computer domain.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

The information in this article applies in the context of Microsoft locator service and the workstations participating in a Windows NT domain model.

Auto handles implicitly make use of the RPC name service calls to locate the server. The locator does a look up for the servers in the local domain only. Thus, if a server is registered in a different domain, calling the remote procedure will create an exception (typically error 1727), assuming there are no RPC servers available that expose the same interface in the local domain.

Additional reference words: 3.51
KBCategory: kbprg kbprb
KBSubcategory: NtwkRpc

PRB: RW1004 Error Due to Unexpected End of File (EOF)

PSS ID Number: Q106064

The information in this article applies to:

- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS
========

The resource compiler generates the following errors when the .RC file includes a .H file whose last line is a define (that is, there was no final carriage return at the end of the #define statement):

```
   fatal error RC1004: unexpected EOF
```

CAUSE
=====

The resource compiler preprocessor follows C syntax. A newline character is required on a #define statement.

RESOLUTION
==========

Add a carriage return following the #define.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbprb
KBSubcategory: TlsRc

PRB: Saving/Loading Bitmaps in .DIB Format on MIPS

PSS ID Number: Q85844

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SYMPTOMS

In Win32, saving or loading a bitmap in .DIB file format is basically the same as in Win16. However, care must be taken in DWORD alignment, especially on the MIPS platform.

An exception occurs when loading or saving a bitmap on the MIPS platform. In NTSD, the following error message is received:

```
data mis-alignment
```

CAUSE

A non-DWORD aligned actual parameter was passed to a function such as GetDIBits().

The .DIB file format contains the BITMAPFILEHEADER followed immediately by the BITMAPINFOHEADER. Notice that the BITMAPFILEHEADER is not DWORD aligned. Thus, the structure that follows it, the BITMAPINFOHEADER, is not on a DWORD boundary. If a pointer to this DWORD misaligned structure is passed to the sixth argument of GetDIBits(), an exception will occur.

RESOLUTION

To resolve this problem, copy the data in the structure over to a DWORD-aligned memory and pass the pointer to the latter structure to the function instead. See the sample code LOADBMP.C for detail.

MORE INFORMATION

The is a sample to illustrates this process. Refer to the LOADBMP.C file in the MANDEL sample that comes with the Win32 SDK.

Additional reference words: 3.10 3.50
KBCategory: kbgraphic kbprb
KBSubcategory: GdiBmp

PRB: Screen Saver Description Not Displayed in Windows 95

PSS ID Number: Q137250

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SYMPTOMS

In Windows 95, when viewing the list of screen savers in the Display Properties dialog box, your screen saver is displayed with the file name instead of the description string.

CAUSE

Windows 95 is identifying the file name as being a long file name because after calling FindFirstFile, it compares the file name to the alternate file name with case sensitivity. If the case on the file names is different, it identifies the file name as a long file name. The alternate file name is always uppercase.

RESOLUTION

There are two solutions. The first and best is to name the file with a long file name that is identical to the description.

The second solution is to rename the file so that the case is the same. The case can be seen using an MS-DOS box. The first file name seen is the alternate file name and the second seen is the long file name. Use the MS-DOS box to rename the file to all uppercase.

NOTE: The descriptive text for screen savers is not taken from IDS_DESCRIPTION as stated in the documentation, but from the module name defined in the .def file. This should be in this form:

```
SCRNSAVE:<decriptive text>
```

Everything after the colon will be included as the descriptive text.

STATUS

This behavior is by design.

Additional reference words: 1.30 4.00
KBCategory: kbui kbprb
KBSubcategory: UsrCtl W32s

PRB: Search Button Disabled in Windows Help

PSS ID Number: Q71761

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

When a help file is loaded into the Windows Help program, the Search button is disabled.

CAUSE

There are two likely causes:

1. The help file defines no keywords for searching.

2. The keywords are defined using a lowercase "k" footnote.

RESOLUTION

For cause 1, if searching is desired, define some keywords in the file. For cause 2, modify the RTF text to use an uppercase "K" for keyword footnotes.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsHlp

PRB: SEH with Abort() in the try Body

PSS ID Number: Q91146

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When using Structured Exception Handling, if the try body calls abort(), the finally body is not executed.

CAUSE

The finally body is not executed because the abort() never returns. It calls ExitProcess(), which terminates the process.

RESOLUTION

This behavior is by design.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg kbprb
KBSubcategory: BseExcept

PRB: SEH with return in the finally Body Preempts Unwind

PSS ID Number: Q91147

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

When using Structured Exception Handling (SEH), returning out of a finally body results in a return from the containing procedure scope. For example, in the following code fragment, the return in the finally block results in a return from func():

```
int func()
{
    int status = 0;
    __try {
       ...
       status = test();
       ...
    }
    __finally {
       if (status != 0) {
          status = FAILURE;
          return status;
       }
    }
    return status;
}
```

CAUSE
=====

A return from within a __finally is equivalent to a goto to the closing brace in the enclosing function [for example, func()]. This is allowed, but has consequences that should normally be avoided.

Exception handling has two stages. First, the exception stack is walked, looking for an accepting __except. When an accepting handler has been found, all __finallys between the top-of-exception-stack and the target __except will be called. During this "unwind", the __finallys are assumed to each execute and then return to their caller (the system unwind code).

A return in a finally abnormally aborts this unwinding. Instead of returning to the system unwinder, the __finally returns to the enclosing function's caller [for example, func()'s parent]. The accepting __except filter may set some status or perform an allocation in anticipation of

the __except handler being entered. In this case, the intervening __finally with the return will stop the unwind, and the __except handler is never entered.

RESOLUTION
==========

This is by design. It makes it possible for a finally handler to stop an unwind and return a status. This is what is referred to as a collided unwind.

Abnormal termination from try/except or try/finally blocks is not generally recommended because it is a performance hit.

The example can be rewritten so that the unwind chain is not aborted:

```
   int func()
   {
      int status = 0;
      __try {
         ...
         status = test();
         ...
      }
      __except(status != 0) {

          /* null */
      }
      if (status != 0)
         status = FAILURE;
      return status;
   }
```

This does not have identical semantics because the exception filters higher up the exception stack will not be executed. However, ensuring that both phases of exception handling progress to the same depth is a more robust solution.

MORE INFORMATION
================

Normally this behavior is transparent to any higher-level exception handling code. If, however, a filter function, as a side effect, stores information that it expects to process in an exception handler, then it may or may not be transparent. Storing such information in a filter function should be avoided because it is always possible that the exception handler will not be executed because the unwind is preempted. In the absence of storing such side effects, it will be transparent that an exception occurred and an attempted unwind occurred if one of the descendent functions has a try/finally block with an finally clause that preempts the unwind.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg kbprb
KBSubcategory: BseExcept

PRB: SelectClipRgn() Cannot Grow Clip Region in WM_PAINT

PSS ID Number: Q118472

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

Setting a smaller clipping region in WM_PAINT by using SelectClipRgn() works fine; however, setting a larger clipping region seems to have no effect. GetClipBox() can be used to verify this after calling SelectClipRgn().

CAUSE

When you call SelectClipRgn() within a BeginPaint()/EndPaint() block in an application's WM_PAINT case, the maximum size to which you can set your clipping region is the size of the update region of your paint structure. This is because the resulting clip region is the intersection of the update region and the region specified in the call to SelectClipRgn(). In other words, you can use SelectClipRgn() to shrink your update region, but not to grow it. This behavior is by design.

RESOLUTION

Invalidate the clipping region area you want before calling BeginPaint(). For example:

```
case WM_PAINT:
   InvalidateRect(hWnd, ....); // Invalidate the size you'll want
                               // for the clip region.
   BeginPaint()
   SelectClipRgn();
   ... paint away ...
   EndPaint();
   break;
```

Something similar could be done in the Microsoft Foundation Classes (MFC), such as:

```
void CMyView::OnPaint()
{
   InvalidateRect(...); // Invalidate the size you'll want.
```

```
   CPaintDC dc(this);   // CPaintDC wraps BeginPaint()/EndPaint().
   // Do drawing here...
}
```

MORE INFORMATION
==========

This is addressed in the documentation for the Windows NT SDK version 3.1 [Section 20.1.5, "Window Regions" in Chapter 20, "Painting and Drawing" in the "Microsoft Win32 Programmer's Reference, Volume 1" or in the Win32 API Reference online help (search on "Window Regions")] which states:

> In addition to the update region, every window has a visible region that defines the window portion visible to the user. The system changes the visible region for the window whenever the window changes size or whenever another window is moved such that it obscures or exposes a portion of the window. Applications cannot change the visible region directly, but Windows automatically uses the visible region to create the clipping region for any display DC retrieved for the window.
>
> The clipping region determines where the system permits drawing. When the application retrieves a display DC using the BeginPaint, GetDC, or GetDCEx function, the system sets the clipping region for the DC to the intersection of the visible region and the update region. Applications can change the clipping region by using functions such as SelectClipPath and SelectClipRgn, to further limit drawing to a particular portion of the update area.

Additional reference words: 3.10 3.50 4.00 SelectClipRegion big small large
KBCategory: kbgraphic kbprb
KBSubcategory: GdiMisc

PRB: Selecting Overlapping Controls in Dialog Editor

PSS ID Number: Q90384

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

Create a dialog box using the Dialog Editor. Place a button onto the dialog box. Create a frame and place it so that it encompasses the button. It is not possible to select that button with the mouse. However, if the frame is created before the button and then moved or placed over the button, then it is possible to select either the frame or the button.

CAUSE

This behavior is by design. When controls are overlapped, the control that is selected when the mouse is clicked is the one that comes last in Z-order.

As a special case, it is possible to select a control placed "underneath" a group box.

RESOLUTION

From the Arrange menu, choose Order/Group. This will bring up a dialog box. Change the Z-order of the button to be after that of the frame. The Z-order may also be changed by manually editing the resource file. The controls that are further down in the file will be "on top."

NOTE: If the frame is selected and is on top of the button, pressing SHIFT+TAB selects the previous control, which will be the button. This does not allow the position of the control to be changed with the mouse; however, it does allow the text and size to be changed.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbprb
KBSubcategory: TlsDlg

PRB: SelectObject() Fails After ImageList_GetImageInfo()

PSS ID Number: Q131279

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

When you try to select the hBitmap returned by ImageList_GetImageInfo() into a device context, the call to SelectObject() fails and returns NULL.

CAUSE
=====

Under the debug version of Windows 95, attempting to select the hBitmap returned by ImageList_GetImageInfo() into a DC causes the GDI to output the message "Bitmap already selected."

Image lists maintain memory DCs with the image and mask bitmaps already selected. This is done to prevent applications from modifying the images contained in the image list that are currently being used by the system. Because a bitmap cannot be selected into more than one DC at a time, applications that call SelectObject() on the same bitmap fail.

RESOLUTION
==========

An application can work around this in Windows 95 by calling CopyImage() on the hBitmap, as demonstrated in the following sample code. This API is new for Windows 95. Remember to delete the hBitmap copy when using this function.

Sample Code

```
HDC        hDC;
HBITMAP    hBitmap, hOldBitmap;
IMAGEINFO  imageInfo;

ImageList_GetImageInfo (hImgList, iWhichImage, &imageInfo);
hBitmap = CopyImage (imageInfo.hbmImage,
                     IMAGE_BITMAP, 0, 0, LR_COPYRETURNORG);

hOldBitmap = SelectObject(hDC, hBitmap);
  :
  :

// Delete hBitmap when you finish using it.
DeleteObject (SelectObject (hDC, hOldBitmap));
```

STATUS
======

This behavior is by design.

Additional reference words: 4.00 unusable hDC Image Lists beta
KBCategory: kbui kbgraphic kbprb kbcode
KBSubcategory: UsrCtl

PRB: SetConsoleOutputCP() Not Functional

PSS ID Number: Q99795

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS

SetConsoleOutputCP() apparently has no effect. The correct codepage is returned from a call to GetConsoleOutputCP, but the displaying of the text remains unchanged.

CAUSE

SetConsoleOutputCP() was designed to change the mapping of the 256 8-bit character values into the glyph set of a fixed-pitch Unicode font, rather than loading a separate, non-Unicode font for each call to SetConsoleOutputCP(). Unfortunately, a fixed-pitch Unicode font was not available by release time, so you can't view the effects of the SetConsoleOutputCP() application programming interface (API) because the currently available console fonts are not Unicode fonts.

STATUS

This behavior is by design in Windows NT versions 3.1 and 3.5.

Additional reference words: 3.10 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseCon

PRB: SetCurrentDirectory Fails on a CD-ROM Drive on Win32s

PSS ID Number: Q125013

The information in this articles applies to:

- Microsoft Win32s, versions 1.0, 1.1, and 1.2

SYMPTOMS

In the following code, assuming that drive E corresponds to a CD-ROM drive, SetCurrentDirectory() always fails to set the current directory to the root directory on the CD-ROM drive. Instead the current directory remains unchanged:

```
   char szCurDir[256];

   SetCurrentDirectory("E:\\");
   GetCurrentDirectory(sizeof(szCurDir),szCurDir);
   MessageBox(NULL, szCurDir, "SCD", MB_OK);
```

CAUSE

SetCurrentDirectory() calls the MS-DOS Interrupt 21h, function 0x4300 to get the file attributes of the specified drive to check whether the specified parameter is a directory. This MS-DOS call always fails if you try to get the attributes of the root directory on a CD-ROM drive, and therefore SetCurrentDirectory() also fails on the root directory of a CD-ROM drive.

STATUS

Microsoft is aware of this problem with SetCurrentDirectory() in Win32s. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

RESOLUTION

As a workaround for the problem with SetCurrentDirectory(), thunk to the 16-bit environment and utilize MS-DOS functions from a 16-bit DLL. For example, you can use Interrupt 21h, function 0x0E (Set Default Drive) followed by Interrupt 21h, function 0x3Bh (Change Current Directory).

MORE INFORMATION

Note that SetCurrentDirectory() fails only on the root directory of a CD-ROM drive. If you pass any directory path other than the root directory to SetCurrentDirectory(), it will work properly.

This is a problem with MS-DOS and can be reproduced from an MS-DOS application in Windows version 3.1.

Additional reference words: 1.00 1.10 1.20 win32s
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: Setup Toolkit File Copy Progress Gauge not Updated

PSS ID Number: Q114609

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

The following may be noticed during the execution of a setup program created with the Setup Toolkit:

1. The copy progress bar appears frozen even though the user may be prompted to change disks.

2. The filename shown at the top of the copy progress dialog box does not change for long periods of time.

CAUSE

The "gas gauge" copy progress dialog box (or Copy Gauge dialog box described on pages 106-107 of the "Setup Toolkit for Windows" manual) is updated only when files are actually being copied to the hard disk. The Setup Toolkit does not update the copy progress dialog box when it checks the version of an existing file. This version check can take a significant amount of time under certain circumstances. A version check is only performed if "Check For Version" is marked in the DSKLAYT program AND the file has a version information resource.

RESOLUTION

There is no way to change this behavior. The dialog box is managed by CopyFilesInCopyList(). The only way to avoid this behavior is to avoid marking files with "Check For Version" in DSKLAYT.

MORE INFORMATION

Under certain circumstances, a file may need to be copied to the temporary directory before its version can be checked. This occurs when the version information in the .INF file matches the version information (exactly) in the file already residing on the hard drive. In this case, the file will be copied from the Setup disks to a temporary location (decompressed if necessary), and other version information will be verified. This can be a time-consuming process and the copy progress dialog box will not be updated while this is occurring.

Additional reference words: 3.10 3.50 4.00 95 MSSETUP CopyFilesInCopyList

KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsMss

PRB: SetWindowsHookEx() Fails to Install Task-Specific Filter

PSS ID Number: Q92659

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

In Windows version 3.1, the SetWindowsHookEx() function fails when it is called to install a task-specific filter (hook) that resides in a DLL.

CAUSE

According to the documentation, the third parameter to the SetWindowsHookEx() function must be the instance handle of the application or the DLL that contains the filter function. However, because of a problem in Windows 3.1, the SetWindowsHookEx() function fails when it is called to install a task-specific filter using the DLL's instance handle.

Note that such a problem does not exist when the SetWindowsHookEx() function is called to install a system-wide filter in a DLL. The DLL's instance handle is accepted as a valid parameter. The first argument passed to the LibMain function of a DLL contains its instance handle.

RESOLUTION

To install a task-specific filter that resides in a DLL, pass the module handle of the DLL as the third parameter to the SetWindowsHookEx() function. The module handle can be retrieved using the GetModuleHandle() function. For example, to install a task-specific keyboard filter, the code might resemble the following:

```
g_hHook = SetWindowsHookEx( WH_KEYBOARD,
                  HookCallbackProc,
                  GetModuleHandle( "HOOK.DLL" ),
                  hTargetTask );
```

This resolution is compatible with future versions of Windows.

Additional reference words: 3.10 3.50 3.51 4.00 95 hook not allowed
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrHks

PRB: ShellExecute() Succeeds But App Window Doesn't Appear

PSS ID Number: Q124133

The information in this article applies to:

- Microsoft Win32s, versions 1.15, 1.15a, and 1.2

SYMPTOMS
========

The following call to ShellExecute() succeeds and the file association is set in File Manager, but the application does not appear to execute (the window is not shown):

```
   hShell = ShellExecute( hWnd,
                          NULL,
                          lpszFile,
                          NULL,
                          lpszDir,
                          SW_SHOWDEFAULT );
```

CAUSE
=====

ShellExecute() is directly thunked to 16-bit Windows. Windows-based applications do not support the SW_SHOWDEFAULT flag.

RESOLUTION
==========

Under Win32s, use SW_NORMAL instead of SW_SHOWDEFAULT when using ShellExecute() with a 16-bit Windows-based application. You can use SW_SHOWDEFAULT if the application specified is a Win32-based application.

STATUS
======

This behavior is by design.

Additional reference words: 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: SHGetNameMappingPtr() and SHGetNameMappingCount()

PSS ID Number: Q142066

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95
- Microsoft Visual C++, 32-bit edition, versions 2.2 and 4.0

SYMPTOMS

Referencing SHGetNameMappingPtr() or SHGetNameMappingCount() in your code generates error C2065: 'DSA_GetItemPtr' : undeclared identifier when building.

CAUSE

These APIs are no longer supported. They being removed from the documentation and headers for future SDK releases.

MORE INFORMATION

The SHFileOperation() API itself is unaffected by this.

Additional reference words: 4.00 2.20 4.40
KBCategory: kbprg kbdocerr kbprb
KBSubcategory:

PRB: Smart Quotes Not in Help Files Compiled From Word 6 RTF

PSS ID Number: Q114604

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

When using the Microsoft Windows Help Compiler (any version) to compile a .RTF file generated by Microsoft Word for Windows version 6.0, any double quotation marks (") in the text may not show up in the compiled help file.

CAUSE

Microsoft Word for Windows 6.0 automatically replaces double quotation marks with "smart quotes". This causes your .RTF file to contain the \ldblquote and \rdblquote tokens rather than the double quotation mark characters. The Help Compiler does not recognize these tokens and therefore ignores them.

RESOLUTION

1. Stop Word for Windows from replacing quotation mark characters in the future by unchecking the "Replace Simple Quotes with Smart Quotes" check box under both of the following dialog boxes:

   a. Choose Options from the Tools menu, then choose Auto Format.

   b. Choose Auto Correct from the Tools menu.

2. Replace existing quotation mark characters by performing a search and replace. That is, search for and replace the smart quotes with the quote character (") (without the parenthesis).

MORE INFORMATION

In addition, apostrophes ('), also known as single quotes, may not show up in the compiled help file. Follow the instructions above to replace the apostrophes as well.

Additional reference words: 3.10 3.50 4.00 95 WINHELP HC HC31 HC30 HCP
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsHlp

PRB: SNMP Extension Agent Gives Exception on Windows NT 3.51

PSS ID Number: Q130562

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.5 and 3.51

SYMPTOMS

An SNMP extension agent built using Windows NT version 3.5 SDK headers and libraries generates an exception when run under Windows NT version 3.51.

For example, the SDK toaster sample (\MSTOOLS\SAMPLES\WIN32\SNMP\TESTDLL) works under Windows NT version 3.5 but generates an exception under Windows NT version 3.51.

CAUSE

The SNMP.LIB SDK library has changed between the release of Windows NT version 3.5 and the release of Windows NT version 3.51. Memory is now allocated dynamically with the Win32 API GlobalAlloc() rather than the c-runtime malloc(). See the SNMP.H SDK header file for details.

An SNMP application that is allocating (or freeing) memory that is passed to a function in SNMP.LIB should use SNMP_malloc() (or SNMP_free()). The sample code for the extension DLL provided with the Windows NT version 3.51 beta SDK incorrectly uses malloc().

RESOLUTION

Rebuild the extension agent with the Win32 SDK headers and libraries for Windows NT version 3.51. Please make sure that the Win32 SDK headers and libraries are used before Visual C++ headers and libraries.

Also, to allocate and free any memory, use the SNMP_malloc() and SNMP_free() macros. Both are defined in SNMP.H.

NOTE: If you are using a beta version of Windows NT version 3.51, please change all references to malloc() and free() in the samples to SNMP_malloc() and SNMP_free(). This is a known problem with the testdll sample (MSTOOLS\SAMPLES\WIN32\WINNT\SNMP\TESTDLL).

STATUS

This behavior is by design.

REFERENCES

For more information, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q124961
TITLE : BUG: SNMP Sample Generates an Application Error

Additional reference words: 3.50
KBCategory: kbnetwork kbprb
KBSubcategory: NtwkSnmp

PRB: SnmpMgrGetTrap() Fails

PSS ID Number: Q130564

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SYMPTOMS

If an SNMP manager application calls SnmpMgrGetTrap() to recieve traps and there are traps available, it returns FALSE. If the application calls GetLastError(), the error code returned is 42 (SNMP_MGMTAPI_TRAP_ERRORS).

RESOLUTION

The SNMP Manager API SnmpMgrGetTrapListen() must be called before calling SnmpMgrGetTrap(). The event handle returned by SnmpMgrGetTrapListen() can then be ignored to poll for traps.

STATUS

This behavior is by design.

REFERENCES

For more information, please see the Microsoft Windows NT SNMP Programmer's Reference (PROGREF.RTF).

Additional reference words: 3.50
KBCategory: kbnetwork kbprb
KBSubcategory: NtwkSnmp

PRB: SnmpMgrStrToOid Assumes Oid Is in Mgmt Subtree

PSS ID Number: Q129063

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SYMPTOMS

When using the SNMP Manager API SnmpMgrStrToOid() and passing it a valid Oid, the application is unable to get the requested variables.

CAUSE

The SNMP Manager API SnmpMgrStrToOid assumes that the Oid that is supplied to it is under the internet MIB of the mgmt subtree (1.3.6.1.2.1.x).

RESOLUTION

To get variables that are not under the mgmt subtree, the Oid must be preceeded by a period (.). For example, say an application is trying to get the system group and the Oid passed to SnmpMgrStrToOid is this:

```
1.3.6.1.2.1.1
```

Then the application will try to get the following, which does not exist:

```
iso.org.dod.internet.mgmt.1.1.3.6.1.2.1.1
```

The correct way to get the system group is to pass this:

```
.1.3.6.1.2.1.1
```

STATUS

This behavior is by design.

REFERENCES

Microsoft Windows/NT SNMP Programmer's Reference (PROGREF.RTF).

Additional reference words: 3.10 3.50
KBCategory: kbnetwork kbprb kbdocerr
KBSubcategory: NtwkSnmp

PRB: Special Characters Missing from Compiled Help File

PSS ID Number: Q86719

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

In version 2.0 of Microsoft Word for Windows, the RTF (rich-text format) source file for a Microsoft Windows Help file contains curly (smart) single or double quotation marks, bullets, or em or en dash characters. However, even though the specified font contains these special characters, they are missing when the compiled file is viewed in Help.

CAUSE

Word for Windows stores special RTF tokens for these characters, which the Microsoft Windows Help Compiler cannot process.

RESOLUTION

In order to avoid these problems do one of the following:

Format the character to use the Symbol font (without using Insert.Symbol).

-or-

Select the desired font (not Symbol) and choose the Symbol option from the Insert menu. Then choose Normal Text in the Symbols From list box. This will format the special character in the desired font.

-or-

Use a bitmap to represent the character or symbol. Use the "bmc" statement to include the bitmap into the text as a character. For more information on the bmc statement, see page 29 of the "Programming Tools" manual provided with version 3.1 of the Microsoft Windows SDK.

Additional reference words: 3.10 3.50 4.00 95 HC HC30.EXE HC31.EXE HCP.EXE
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsHlp

PRB: Spy Repeatedly Lists a Single Message

PSS ID Number: Q74278

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

SPY sometimes reports messages as having been sent twice. Messages from DDE conversations, especially the WM_DDE_DATA, WM_DDE_ACK, and WM_DDE_POKE messages, are the most often duplicated. This behavior can be seen by choosing All Windows from Spy's Windows menu, selecting DDE messages in Spy's Options dialog, and then running two applications that communicate using DDE.

CAUSE

Spy displays a message each time it is retrieved. If an application retrieves a message once by calling PeekMessage() with PM_NOREMOVE, and then again with GetMessage(), Spy will report it twice. Spy cannot determine that the message was already retrieved by the application. Since the application is using the message twice, the message should indeed be shown twice. This is a useful feature for determining how an application is handling the DDE messages sent to it.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsSpy

PRB: StartDoc() Fails with Non-Zeroed DOCINFO

PSS ID Number: Q135119

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

A call to StartDoc() fails, and no other reason for failure can
be found.

CAUSE
=====

The DOCINFO structure passed to StartDoc() is not initialized to zeros
before use.

RESOLUTION
==========

Perform the following three steps:

1. Call memset() to initialize the structure to zeros.

2. Set the cbSize member to the appropriate value.

3. Set the other relevant structure members' values.

Sample Code

Here is an example of what this code might be:

```
DOCINFO   di;

// Get the DC, SetAbortProc(), and so on.

memset( &di, 0, sizeof( DOCINFO ) );
di.cbSize = sizeof( DOCINFO );
di.lpszDocName = "MyDoc";
if( StartDoc( hDC, &di ) <= 0 )
   HandleFailure();
```

MORE INFORMATION
================

As a general rule, any structure that has a member that indicates the size
of the structure should be initialized to all zeros before being used by

following the previous steps.

Additional reference words: 3.10 3.50 4.00 fail error print DC
KBCategory: kbprint kbprb kbcode
KBSubcategory: GdiPrn

PRB: Starting a Service Returns "Logon Failure" Error

PSS ID Number: Q120556

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS

Starting a service from either the service control manager or from the StartService API may return error 1069, "ERROR_SERVICE_LOGON_FAILED."

CAUSE

This error occurs if the service was started from an account that does not have the "Log on a service" privilege.

RESOLUTION

An account can be granted the "Log on a service right" through the User Manager Application. From the Policies menu, choose User Rights. In the User Rights Dialog Box, select the "Show Advanced User Rights" check box. Choose "Log on a service," in the "Right" scroll box. Then choose the Add button to grant your account the "Log on a service" privilege.

Additional reference words: 3.10 3.50
KBCategory: kbprg kbprb
KBSubcategory: BseService

PRB: Successful LoadResource of Metafile Yields Random Data

PSS ID Number: Q86429

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When an application for the Microsoft Windows graphical environment calls the LoadResource() function to load a metafile from the application's resources, locks the metafile with the LockResource() function, and uses the metafile, the application receives random data even though the LoadResource() and LockResource() functions indicate successful completion.

CAUSE

The application loaded the metafile previously and when the application freed the metafile, it used the DeleteMetaFile() function to invalidate the metafile handle.

RESOLUTION

Modify the code that unloads the metafile from memory to call the FreeResource() function.

MORE INFORMATION

The LoadResource() and FreeResource() functions change the lock count for a memory block that contains the resource. If the application calls DeleteMetaFile(), Windows does not change the lock count. When the application subsequently calls LoadResource() for the metafile, Windows does not load the metafile because the lock count indicates that it remains in memory. However, the returned memory handle points to the random contents of that memory block.

For more information on the resource lock count, query in the Microsoft Knowledge Base on the following words:

   multiple and references and LoadResource

Most of the time, an application uses the DeleteMetaFile() function to remove a metafile from memory. This function is appropriate for

metafiles created with the CopyMetaFile() or CreateMetaFile()
functions, or metafiles loaded from disk with the GetMetaFile()
function. However, DeleteMetaFile() does not decrement the lock count
of a metafile loaded as a resource.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 EnumMetaFile
GetMetaFile GetMetaFileBits PlayMetaFile PlayMetaFileRecord SetMetaFileBits
SetMetaFileBitsBetter
KBCategory: kbui kbprb
KBSubcategory: UsrRsc

PRB: TAB Key, Mnemonics with FindText and ReplaceText Dialogs

PSS ID Number: Q96134

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

When implementing the FindText and/or ReplaceText common dialog box, the dialog box is displayed but the TAB key and mnemonics do not work properly.

CAUSE
=====

The FindText and ReplaceText common dialog boxes are modeless dialog boxes. Therefore, a call to IsDialogMessage must be made in the application's main message loop in order for the TAB key and mnemonics to work properly.

RESOLUTION
==========

This problem can be corrected by adding a call to IsDialogMessage() in the application's main message loop.

MORE INFORMATION
================

A typical message loop might resemble the following

```
   while (GetMessage(&msg,NULL,NULL,NULL))
     {
        if ( ghFFRDlg==NULL || !IsDialogMessage(ghFFRDlg, &msg) )
           {
              TranslateMessage(&msg);
              DispatchMessage(&msg);
           }
     }
```

where ghFFRDlg is a global window handle for the currently active modeless common dialog box.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrCmnDlg

PRB: Tape Variable Blocksize Limited to 64K Bytes

PSS ID Number: Q152518

The information in this article applies to:

- Microsoft Win32 Application Programming Interface, included with: Microsoft Windows NT, versions 3.51, 4.0

SYMPTOMS

When using the Variable Blocksize technique of writing data to a tape device on Windows NT, block sizes may be limited to either 64K or 128K bytes.

CAUSE

SCSI Tape devices can only write to each tape block the number of bytes transferable over the SCSI Bus in a single I/O operation. Each Host Bus Adapter (HBA)-specific SCSI miniport driver specifies the maximum number of pages that the HBA can scatter/gather for the DMA transfer. Multiplying the page size times the number of scatter/gather entries will yield the effective limit to the number of bytes that can be transmitted in a single I/O operation.

On x86 Windows NT, PAGE_SIZE is 4K bytes and many popular SCSI HBAs (including Adaptec xx40) support a maximum of 16-17 pages, yielding a limit of 64K bytes transferred during a single I/O operation.

Some BusLogic HBAs support 31 scatter/gather entries yielding slightly less than 128K bytes per I/O.

STATUS

This behavior is by design.

Additional reference words: 3.51 4.00
KBCategory: kbprg kbprb
KBSubcategory: bseFileio

PRB: Taskbar Anomolies When Application Larger than the Screen

PSS ID Number: Q142166

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95

SYMPTOMS

If your application is sized larger than the available screen size, the taskbar does not act consistantly with respect to staying in the forground or dropping to the background when the "Always on Top" setting is on.

Basically, if an application is larger than the screen resolution and the left edge of the app is visible, the taskbar remains topmost once it has been touched. If the left edge of the application is not visible, the taskbar continues to fall behind the application each time it gets the focus.

RESOLUTION

Make sure when sizing your window to use GetSystemMetrics() with either SM_C[X|Y]MAXIMIZED or SM_C[X|Y]FULLSCREEN as appropriate to get an appropriate size for your window. Do not exceed the size specified by SM_C[X|Y]SCREEN. This is especially important when responding to WM_DISPLAYCHANGE messages.

STATUS

This behavior is by design.

Additional reference words: 4.00
KBCategory: kbui kbprb
KBSubcategory:

PRB: time() Incorrect If TZ Variable Not Defined

PSS ID Number: Q149702

The information in this article applies to:

- Microsoft Win32s versions 1.30, 1.30c

SYMPTOMS
========

When you call the C run-time function, ctime() or localtime() from a 32-bit applicaiton running under Win32s and the TZ environment variable is not set to zero, the displayed time is incorrect. The following piece of code demonstrates the problem.

```
#include <windows.h>
#include <stdio.h>
#include <time.h>

   time_t   timet;
   char  szMsg[256];

   time(&timet);
   wsprintf(szMsg, "The time is %s\n", ctime(&timet));
   MessageBox(NULL, szMsg, "Win32s Test - time() function", MB_OK);
```

CAUSE
=====

The function that causes the time shift is ctime(), not time(). Actually ctime() calls localtime(), which subtracts nine hours by default if the TZ environment variable is not defined. If TZ is set to zero, the time will be displayed correctly.

STATUS
======

This behavior is by design.

Additional reference words: 1.30 1.30c kbinf crt time
KBCategory: kbprg kbprb
KBSubcategory: w32s

PRB: TrackPopupMenu() on LoadMenuIndirect() Menu Causes UAE

PSS ID Number: Q75254

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

When LoadMenuIndirect() is used to create a menu from a menu template and the menu handle is passed to TrackPopupMenu(), Windows reports an unrecoverable application error (UAE). Windows NT and Windows 95 display the floating pop-up menu as a vertical bar.

CAUSE

The menu handle returned from LoadMenuIndirect() does not point to a menu with the MF_POPUP bit set.

RESOLUTION

The following code fragment demonstrates the correct procedure to "wrap" the menu created by LoadMenuIndirect() inside another menu. This procedure sets the MF_POPUP bit properly.

```
   hMenu1 = LoadMenuIndirect(lpMenuTemplate);

   hMenuDummy = CreateMenu();
   InsertMenu(hMenuDummy, 0, MF_POPUP, hMenu1, NULL);

   hMenuToUse = GetSubMenu(hMenuDummy, 0);
```

Use hMenuToUse when TrackPopupMenu() is called. The values of hMenu1 and hMenuToUse should be the same.

When the menu is no longer required, call DestroyMenu() to remove hMenuDummy. This call will also destroy hMenu1 and free the resources it used.

Additional reference words: 3.00 3.10 3.50 4.00 gpf
KBCategory: kbui kbprb kbcode
KBSubcategory: UsrMen

PRB: TransactNamedPipe() Returns Error 230

PSS ID Number: Q140022

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51

SYMPTOMS

When you use Named Pipes and run the client and server on the same computer, if TransactNamedPipe() is called from the client side, the error code returned is 230 (the pipe state is invalid).

CAUSE

The client side is running in byte mode as opposed to message mode as required by TransactNamedPipe().

RESOLUTION

Use SetNamedPipeHandleState() to set the pipe mode to message mode on the client side.

STATUS

This behavior is by design.

MORE INFORMATION

If the client and server are running on the same computer, it is possible to open the client side without explicitly specifying the computer name using this format:

```
\\.\pipe\test
```

This is processed by the Named Pipe File System (NPFS), which defaults to byte mode. For this reason, if SetNamedPipeClientHandle() is not called on the client side, calling TransactNamedPipe() will fail with the error 230.

Instead of using the previous format, a client application running on the same computer (or on a different computer) can use the computer name as shown here:

```
\\Server\pipe\test
```

When the computer name is used like this, the request is processed through the redirector, which defaults to message mode. Therefore, a call to SetNamedPipeHandleState() is not required.

Please note that a robust application should call SetNamedPipeHandleState() even if the client is running on a different computer or using the computer name if running on the same computer.

Additional reference words: 3.50 3.51
KBCategory: kbnetwork kbdocerr kbprb
KBSubcategory: NtwkMisc

PRB: Trouble Enumerating a Remote Server's Ports in Windows 95

PSS ID Number: Q150522

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95

SYMPTOMS

In Windows 95, when a valid remote server name is specified in the first parameter, the EnumPorts API fails and GetLastError reports error 0x87 (Invalid Parameter). The EnumPorts API succeeds in enumerating local ports when NULL is passed as the first parameter.

CAUSE

By design, the built-in print providers in Windows 95 do not support remote enumeration of ports.

RESOLUTION

Alternate print providers can be installed that do support remote enumeration of ports. For information on creation and installation of print providers in Windows 95, see the Windows 95 DDK.

STATUS

This behavior is by design.

Additional reference words: 4.0 remote network list port permission
KBCategory: kbgraphic kbprb
KBSubcategory: GdiPrint

PRB: Trouble Using DIBSection as a Monochrome Mask

PSS ID Number: Q149585

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for:
  - Windows NT versions 3.5 and 3.51
  - Windows 95

SYMPTOMS

When you use BitBlt() to create a monochrome mask on a 1bpp DIBSection, target pixel color is chosen without regard for the background color in the target device context. The AND mask that results is not what the programmer intended, and subsequent transparent blts using that mask do not maintain proper transparency.

CAUSE

BitBlt() with a DIBSection as a target results in color matching without regard for the current background color. This means that GDI decides whether black or white would be a closer match for each color in the bitmap and sets the target pixel accordingly.

RESOLUTION

Either use a Device Dependent Bitmap (DDB) for the mask, or set the pixels in the DIBSection manually. To perform this task manually would require checking each pixel against the background color and setting the target pixel to white for those that match.

STATUS

This behavior is by design.

MORE INFORMATION

A commonly used method for creating an AND mask for transparent BitBlts involves blting the color source image to a monochrome bitmap. When converting color to monochrome, BitBlt() uses the background color (set with SetBkColor()) to determine which pixels should be white and which should be black. However, this is true only for device dependent bitmaps. When the target of a BitBlt() is a DIBSection, color matching is always performed. This means that, for a DIBSection, the background color is ignored, and colors in the source image are matched to white or black based on color matching.

Additional reference words: 3.51 3.50 4.00 CreateDIBSection StretchBlt
BitBlt mono transparency
KBCategory: kbgraphic kbprb
KBSubcategory: GdiBmp

PRB: Unable to Choose Kanji Font Using CreateFontIndirect

PSS ID Number: Q119914

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

You have difficulty getting the Japanese font handle in Microsoft Win32 Software Development Kit (SDK) for Japanese Windows 95. If you use CreateFontIndirect() to create the font handle, only the English font is selected. You are unable to select Kanji (the main system of writing in Japan) fonts such as MS Mincho and MS Gothic.

CAUSE
=====

The lfCharSet field in the LOGFONT structure is not set to "SHIFTJIS_CHARSET".

RESOLUTION
==========

Specify SHIFTJIS_CHARSET as the value for lfCharSet field.

MORE INFORMATION
================

In Windows 95, Japanese fonts cannot be selected without the lfCharSet field being set to SHIFTJIS_CHARSET. This standard was not enforced in Japanese Windows NT and Japanese Windows 3.1, So an application with lfCharSet set to a value other than SHIFTJIS_CHARSET might be able to select Japanese fonts under Japanese Windows NT and Japanese Windows version 3.1 using CreateFontIndirect(), but not under Japanese Windows 95.

ShiftJIS is a double-byte character set (DBCS) unique to the Japanese version of Windows NT, Windows 95, and Windows version 3.1. It requires a specialized font, keyboard input, and DBCS string-handling support.

Additional reference words: 4.00 kbprb
KBCategory: kbgraphic kbprb
KBSubcategory: GdiFnt WIntlDev

PRB: Unicode ChooseColor() Help Button May Crash Common Dialog

PSS ID Number: Q102026

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1, 3.5, and 3.51

SYMPTOMS

A Unicode compiled program, or a program that explicitly calls the ChooseColorW() function, may suffer an access violation in the common dialog boxes if the Help button in the ChooseColor() dialog box is chosen.

CAUSE

This problem may not be 100-percent reproducible. Because of the way that the program loader loads code into memory, a certain address referenced by the common dialog box Help button logic may or may not be valid depending on what other applications have been run, what application is calling ChooseColorW(), and the order in which these programs are run.

WORKAROUND

The following are suggested workarounds:

- Try running an ANSI-compiled application that uses the common dialog boxes, and open one of the common dialog boxes before you run the application having this problem.

- A workaround for new applications developers is to call ChooseColorW() with a hook function (see the Windows Software Development Kit documentation for more information on common dialog box hook functions), handle the Help button logic, and return TRUE from the hook function to bypass the standard common dialog box push-button code.

Additional reference words: 3.10 3.50
KBCategory: kbprg kbprb
KBSubCategory: UsrCmnDlg

PRB: UnrealizeObject() Causes Unexpected Palette Behavior

PSS ID Number: Q86800

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

In an application for the Microsoft Windows graphical environment, when a logical palette (HPALETTE) is used with a device-dependent bitmap (DDB) and the application realizes the palette, the DDB is painted in incorrect colors.

CAUSE

The UnrealizeObject function was previously called to unrealize the palette.

RESOLUTION

Modify the code to remove the call to UnrealizeObject()/

MORE INFORMATION

Because the colors of a DDB are stored using indices into the system palette rather than explicit RGB colors, proper DDB rendering depends on the colors of the system palette being set properly. An application sets up the system palette to display a DDB by realizing a logical palette. The realization process changes the colors in the system palette and creates a mapping between entries in the logical palette and entries in the system palette.

When an application first renders a logical palette with RealizePalette(), Windows sets an internal flag to indicate that the palette has been realized and stores the current mapping from logical palette entries to physical palette entries. When an application realizes the palette again (for example, after another application modifies the palette), Windows restores the effected entries of the system palette to the state they had when the logical palette was realized for the first time.

This mechanism allows a bitmap first realized with a specific palette to display correctly when the same palette is realized subsequently.

If the application calls UnrealizePalette() on a logical palette, Windows discards the stored state information for the palette. If the application realizes the palette subsequently, its colors may map into new locations in the system palette. Because the bitmap contains indices into the old system palette, it may display incorrectly.

To address this situation, do not call UnrealizeObject() on a palette if the application has a DDB that depends on that palette.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic kbprb
KBSubcategory: GdiPal

PRB: Using Animation Resources with the Animation Control

PSS ID Number: Q149688

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0
- Microsoft Win32s version 1.3

SYMPTOMS
========

When sending the ACM_OPEN message to an animation control, you have the option of using an ANI file or an ANI resource. When you specify a resource, you cannot specify which module to load that resource from.

CAUSE
=====

When loading an animation resource, the control uses the hInstance that it was created with (the hInstance passed to CreateWindow, DialogBox, or CreateDialog).

RESOLUTION
==========

Use an animation file or store the animation resource in the same module from which the animation control was created. One other alternative, which is probably not a good idea, is to retrieve the resource by using FindResource, LoadResource, and LockResource, and then save that to a temporary animation file. The drawback to this method is that you need to make sure that the temporary file is deleted.

STATUS
======

This behavior is by design.

Additional reference words: 1.30 4.00
KBCategory: kbprg kbui kbprb
KBSubcategory: UsrCtl w32s

PRB: VerLanguageNameA() Not Exported in Win32s

PSS ID Number: Q147876

The information in this article applies to:

- Microsoft Win32s version 1.3c

SYMPTOMS
========

If you call VerLanguageName() from a 32-bit application running under Win32s, you will get an Undefined link error on VerLanguageNameA in Version.dll.

CAUSE
=====

In Win32s, Version.dll does not export this function. In Windows NT and Windows 95, VerLanguageNameA is exported in both Version.dll and Kernel32.dll. In Windows NT, Version.dll forwards the export of VerLanguageName() to the export in Kernel32.dll. In Windows 95, Version.dll separately calls the function in Kernel32.dll.

RESOLUTION
==========

Under Win32s, you can still use the VerLanguageName() function from your application because the function exists in Kernel32.dll. You will need to specify Kernel32.lib before Version.lib on your link line. Then your application will use the function from Kernel32.dll.

STATUS
======

This behavior is by design.

Additional reference words: 1.30c win32s gdi
KBCategory: kbprg kbprb
KBSubcategory: w32s kbgdi

PRB: Vertical Scroll Bars Missing from Windows Help

PSS ID Number: Q77841

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

The vertical scroll bars do not appear in Windows Help when the window is sized smaller than the amount of text being displayed.

CAUSE

The displayed topic is formatted as a nonscrolling region.

RESOLUTION

Select the topic in the RTF editor and turn off the "Keep Paragraph with Next" formatting. This format is used by the Help Compiler to delimit the nonscrolling regions.

MORE INFORMATION

To remedy the situation in Microsoft Word for Windows, perform the following three steps:

1. Highlight the entire topic.

2. From the Format menu, choose Paragraph.

3. In the Format Paragraph dialog box, cancel the "With Next" check box.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool kbprg kbprb
KBSubcategory: TlsHlp

PRB: Video for Windows Skips AVI Frames

PSS ID Number: Q122775

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SYMPTOMS

When Video For Windows is playing an AVI file, the processing power of the computer system and the data rate of the AVI stream determine whether frames will be skipped during playback. If the data rate exceeds the ability of Video For Windows to render all the AVI frames in time, Video For Windows will first try to catch up by decompressing delta frames but not drawing them. If Video For Windows still can't catch up, it will eventually skip all the way to the next key frame before it renders a new frame.

STATUS

This behavior of Video for Windows is by design and cannot be controlled by the application being affected.

Additional reference words: 3.10 3.50 4.00 95 VfW AVI
KBCategory: kbmm kbprb
KBSubcategory: MMVideo

PRB: Video Window Moves During Playback

PSS ID Number: Q139977

The information in this article applies to:

- Microsoft Video for Windows Development Kit version 1.1
- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 4.0

SYMPTOMS

The video playback window shifts position at the start of playback.

CAUSE

Microsoft Windows versions 3.1 and 3.11 along with Windows 95 automatically position the top-left corner (origin) of the video image (client area) at specific screen coordinates for playback performance reasons. Therefore, playback in a non-aligned client area can be slowed by up to 50%.

RESOLUTION

To avoid the window repositioning, an application must set the top-left coordinates of the window's client area in a correctly-aligned screen location prior to playback. Select the nearest x and y screen coordinates for the client area that are evenly divisible by 4. For example, use (12,12) instead of (13,13), and use (20,16) for (19,16) or (20,17).

There are no flags that disable the automatic alignment.

STATUS

This behavior is by design.

MORE INFORMATION

To provide optimal playback across the broad spectrum of video boards and displays supported by Windows, the origin of the video image is aligned along a 32-bit boundary. Client areas not aligned along 32-bit boundaries will cause the playback windows to be repositioned towards the upper-left corner of the screen. Aligning the video image on a four-pixel boundary helps achieve the highest frames per second.

Stretching playback may also slow performance down by 50%. In most cases, you should play the movie in its standard format unless stretching is required. For example, play a 320x240 movie at 320x240.

Additional reference words: 3.10 4.00 3.50 alignment avi move vfw pixel

pixels
KBCategory: kbmm kbprg kbprb
KBSubcategory: MMVideo

PRB: Why Thunking to 16-bit MAPI Will Fail Under Win32s

PSS ID Number: Q149710

The information in this article applies to:

- Microsoft Win32s version 1.30, 1.30c

SUMMARY
=======

This article discusses the details on why a Win32s client (a 32-bit thunk DLL) cannot thunk to the 16-bit MAPI DLL. It also suggests several workarounds for this situation.

SYMPTOMS
========

The MAPI interface is not supported under any version of Win32s. In addition, a Win32s client (a 32-bit application or a DLL) cannot directly thunk down to 16-bit MAPI APIs in Windows 3.x. There are several resolutions for this scenario.

CAUSE
=====

When the 16-bit MAPI DLL allocates a buffer on behalf of a call from the 32-bit thunk DLL, your 16-bit code will have to call UTSelectorOffsetToLinear() on the returned 16:16 address. The result is a flat 32-bit address that can then be passed back to the 32-bit DLL. But when the 32-bit code is ready to call the 16-bit MAPIFreeBuffer() function by way of the thunk, the flat 32-bit address needs to be converted back to the original 16:16 address that the 16-bit code can use. The normal solution is that the 16-bit code would call UTLinearToSelectorOffset() on the 32-bit address passed from the 32-bit client before making a call to MAPIFreeBuffer().

But the problem is that in the following nested function call where x is a segmented (16:16) address, Win32s does not guarantee to return the original value of x (16:16 address) back. Hence, when MAPIFreeBuffer is called with this bogus 16:16 address, it fails.

```
   UTLinearToSelectorOffset( UTSelectorOffsetToLinear(x) )
```

Note that this address translation problem described here is not specific to the MAPI allocation and de-allocation routines. It applies to all 16-bit MAPI functions that use 16:16 segmented addresses.

RESOLUTION
==========

Here are three approaches you can use to resolve this situation:

- Correct the address translation problem that is inherent to Win32s. For

each 16-bit MAPI function that would use or return a 16:16 segmented address and that needs to be converted to a 32-bit address to be passed on to the 32-bit side, you need to maintain a global lookup table or array on the 16-bit side that associates the new 32-bit address with the corresponding 16:16 address. When the 32-bit code calls into 16-bit code passing a valid 32-bit address, instead of calling UTLinearToSelectorOffset() on this 32-bit address, you will have to fetch the corresponding 16:16 address from the lookup table. This 16:16 address can be used with all the MAPI functions on the 16-bit side.

- Write a separate 16-bit client (a 16-bit DLL) that would implement the MAPI functionality by linking to the 16-bit MAPI DLL. The Win32s client (a 32-bit DLL) would then thunk to the 16-bit client DLL that makes the MAPI calls. The address translation problem does not exist in this scenario because all MAPI functionality is implemented and executed on the 16-bit side by the 16-bit client DLL linking to the 16-bit MAPI DLL.

- Write a separate 16-bit client (a 16-bit application) that implements the MAPI functionality by linking to the 16-bit MAPI DLL. Then the Win32s client (either a 32-bit application or DLL) can communicate with this 16-bit client by way of the documented IPC (interprocess communication) methods that are available to Win32s. For more information on IPC mechanisms under Win32s, please see the following article in the Microsoft Knowledge Base:

  ARTICLE-ID: Q95900
  TITLE : Interprocess Communication on Windows NT, Windows 95, & Win32s

Note that Visual Foxpro version 3.0b includes a DLL called Foxmapi.fll that implements the second workaround, allowing 32-bit Visual Foxpro applications to communicate with 16-bit MAPI functions under Win32s.

These three approaches are only suggestions; Microsoft cannot provide support if you decide to implement them in your Win32s-based application.

STATUS
======

This behavior is by design.

Additional reference words: 1.30 1.30c kbinf ipc mapi thunk win16 win32
KBCategory: kbprg kbprb
KBSubcategory: w32s

PRB: Win32-Based Screen Saver Shows File Name in Control Panel

PSS ID Number: Q126239

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SYMPTOMS

After writing a Win32-based screen saver, if the IDS_DESCRIPTION string is missing from the string table, the file name of the screen saver module is displayed in the desktop control panel applet. This happens even if the DESCRIPTION entry is specified in the .DEF file of the module.

CAUSE

Each Windows screen saver has a name, which is a string packed into the screen saver module by the linker. This name is displayed in the screen saver name drop down list box under the desktop applet in the control panel. The user can select different screen savers for the desktop through this list box.

Under 16-bit Windows, the name of a screen saver is specified by the DESCRIPTION entry in the .DEF file. Under Windows NT, this is no longer true. Instead, a special entry in the string table is used to specify the name of a screen saver. This string must have IDS_DESCRIPTION as its string ID. IDS_DESCRIPTION is defined in SCRNSAVE.H.

If the DESCRIPTION entry is missing from the .DEF file for a 16-bit screen saver, or the IDS_DESCRIPTION is missing from the string table for a 32-bit screen saver, Windows NT displays the file name of the screen saver module in the drop down list box in the Desktop control panel applet.

RESOLUTION

Place the DESCRIPTION entry in .DEF file for a 16-bit screen saver, or place the IDS_DESCRIPTION in the string table for a 32-bit screen saver.

STATUS

This behavior is by design.

REFERENCES

Chapter 14, "Screen Saver Library," Microsoft Windows Software Development Kit, version 3.1, Programmer's Reference, Volume 1: Overview.

Chapter 79, "Screen Saver Library," Microsoft Win32 Programmer's Reference, Volume 2: System Services, Multimedia, Extensions, and Application Notes.

Additional reference words: 3.10 3.50
KBCategory: kbgraphic kbprb
KBSubcategory: GdiScrsav

PRB: Win32s GetVolumeInformation() Returns 0x12345678 or 0

PSS ID Number: Q93639

The information in this article applies to:

- Microsoft Win32s version 1.0, 1.1, 1.15, and 1.2

SYMPTOMS
========

In Win32s version 1.0 and 1.1, GetVolumeInformation() always returns a volume ID of 0x12345678. In Win32s version 1.15 and later, the return is 0.

STATUS
======

This is a known limitation of Win32s and is by design.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: Win32s Limits DDE Atom Strings

PSS ID Number: Q147847

The information in this article applies to:

- Microsoft Win32s version 1.3

SYMPTOMS
========

Win32s limits the length of DDE atom strings to 80 characters. If a string of more than 80 characters is specified, the string will be truncated and you won't be able match your atoms to those supplied with DDE messages.

CAUSE
=====

By design, this is a Win32s limit. No change to Win32s on this matter is likely.

RESOLUTION
==========

The only way to avoid this behavior is to keep your DDE names strings to less than 80 characters.

STATUS
======

This behavior is by design.

Additional reference words: 1.30
KBCategory: kbprg kbui
KBSubcategory: UsrDde W32s

PRB: Windows 95 SCSI Miniport Drivers May Generate Bad CDBs

PSS ID Number: Q141205

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SYMPTOMS
========

Windows 95 SCSI Miniport drivers may generate ATAPI 12-byte CDBs that contain trailing random characters.

CAUSE
=====

Windows 95 SCSI Miniport drivers that handle ATAPI commands often need to convert the SCSI-2 6-byte or 10-byte CDBs sent by Windows 95 into 12-byte CDBs for their ATAPI hardware. The Input Output Supervisor (IOS) under Windows 95 does not zero out the CDB array in the SCSI Request Block (SRB) before writing data to it. If a driver handling ATAPI commands does not zero out the last unused bytes in the 12-byte CDB before sending the CDB to the controller, unpredictable results may occur.

RESOLUTION
==========

Zero out the unused bytes of the 12-byte CDB before passing it to the controller.

Additional reference words: 4.00 garbage
KBCategory: kbprg kbprb
KBSubcategory:

PRB: Windows NT 3.50 Fonts Look Different from v.3.51 Fonts

PSS ID Number: Q137328

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51 and 4.0
- Microsoft Windows NT version 3.51

SYMPTOMS

An application can instruct Windows that a fixed (non-proportional) system font should be used for a dialog box by specifying "Fixedsys" in the dialog template in the applications resource file (.RC file). For example:

```
IDD_ABOUTBOX DIALOG DISCARDABLE  22, 17, 167, 64
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "About Generic"
FONT 9, "Fixedsys"
BEGIN
...
```

The fixed system font used in a dialog box in Windows NT version 3.50 differs from that used in version 3.51. A 32-bit Windows-based application run under Windows NT version 3.50 displays a bolder looking dialog box font. In addition, the font editor in Microsoft Visual C++ displays the system fixed font as it would look under Windows NT version 3.50, so the font is incorrectly displayed for a Windows NT version 3.51 application.

RESOLUTION

To instruct Windows to use the bolder font in Windows NT version 3.51 as it does in version 3.50, you need to mark the application as a version 3.10 application. This will also ensure that the application's dialog box font looks the same under the Visual C++ dialog editor as it does when run under Windows NT 3.51. To mark your application as version 3.10, click Settings on the Project menu in Visual C++, click the Link tab in the Project Options dialog box, and add this:

```
    /SUBSYSTEM:windows,3.10
```

You can also add this manually to the "linker" options in your makefile.

Additional reference words: 4.00
KBCategory: kbgraphic kbprb
KBSubcategory: GdiFnt

PRB: Windows Properties Warning Using GWL_USERDATA in Win32s

PSS ID Number: Q147433

The information in this article applies to:

 - Microsoft Win32s version 1.3

SYMPTOMS
========

If an application uses SetWindowLong() with GWL_USERDATA in Win32s, then the debug version of Windows will issue warning messages saying that some window properties were not removed when the window is destroyed.

CAUSE
=====

GWL_USERDATA is supported under Win32s, even though this feature is not available under Windows 3.x.

In Win32s, GWL_USERDATA is implemented by adding two window properties to the window, one per 16-bit property. These window properties are not removed by Win32s when the window is destroyed.

RESOLUTION
==========

Even though Win32s does not remove the window properties, they are actually deleted by Windows upon application termination. No memory leak will occur from this problem, so it is safe to ignore the warning in most cases.

In the case where your application uses the GWL_USERDATA with a large number of windows, these window properties are not freed until the application terminates. A workaround or suggestion for such a situation would be to use WNDCLASS.cbWndExtra instead of GWL_USERDATA.

STATUS
======

This behavior is by design.

Additional reference words: 1.30 win32s gwl_userdata
KBCategory: kbprg kbprb
KBSubcategory: w32s

PRB: Windows REQUEST Function Not Working With Excel

PSS ID Number: Q26234

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

The REQUEST function does not work correctly with Excel. The request message is received, however, Excel does not process the WM_DDE_DATA message that is sent back.

RESOLUTION

The fResponse bit must also be set in the WM_DDE_DATA message (bit 12). This bit tells Excel that the data message is in reply to a REQUEST function and not an ADVISE function. If "lpddeup->fResponse=1" is added, the REQUEST function should work correctly.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrDde

PRB: WinExec() Fails Due to Memory Not Deallocated

PSS ID Number: Q126710

The information in this article applies to:

- Microsoft Win32s, versions 1.15, 1.2, and 1.25

SYMPTOMS

Under Win32s version 1.15, when a Win32-based application spawns a 16-bit application several times using WinExec(), after a few successful spawns, WinExec() fails.

Each time WinExec() is called to start a 16-bit application, Win32s allocates a fixed and pagelocked block. The owner of this block is the Win32-based application. The memory is not deallocated when the 16-bit application is terminated, only when the 32-bit application is terminated.

CAUSE

This is actually a bug in Windows version 3.1. The 32-bit WinExec() calls the 16-bit LoadModule(). Win32s passes the environment of the calling process to LoadModule(). Then the Windows 3.1 LoadModule() allocates a buffer for the environment, copies this environment to the buffer, and passes this buffer to the child process. The problem is that the owner of the new allocated buffer is the parent, so the memory is freed when the parent exits. There is no code for otherwise freeing the memory. This bug also affects 16-bit Windows-based applications if LoadModule() is called with an environment selector that is not NULL.

In a related problem, when the parent terminates, the child's environment becomes invalid. This may cause a general protection (GP) fault.

RESOLUTION

To work around the problem, you can call the Windows version 3.1 WinExec() through the Universal Thunk. However, the parent will not be able to modify the child's environment.

In Win32s version 1.2x, this problem exists only if you start 16-bit application using CreateProcess() or LoadModule() and pass it explicit environment strings. In this case, you will encounter the Windows version 3.1 bug. If you do not pass an explicit environment, the environment passed to the 16-bit application is NULL. This resolves the problems mentioned in the Symtpoms and Cause sections of this article.

MORE INFORMATION

With the changes in Win32s version 1.2x, if the calling application modifies the environment, the child process will not get the modified environment of the parent. It will get the global MS-DOS environment. This is also true for WinExec().

If you need to pass a modified environment, call LoadModule() or CreateProcess() with the environment set to what GetEnvironmentStrings() returns. Be aware that this will cause a memory leak. In addition, if the parent terminates before the child, the child's environment will become invalid.

Additional reference words: 1.20 GPF
KBCategory: kbprg kbprb
KBSubcategory: W32s

PRB: WINS.MIB & DHCP.MIB Files Missing from Win32 SDK 3.5

PSS ID Number: Q121625

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SYMPTOMS

Two Simple Network Management Protocol (SNMP) files (WINS.MIB and DHCP.MIB) are missing from the currently released version of the Win32 SDK for Windows NT version 3.5.

The files are two Management Information Base (MIB) files for WINS and DHCP SNMP usage. The two files (WINS.MIB and DHCP.MIB) are used in the generation of a MIB.BIN file for use with the NT SNMP Extendible Agent Management API.

RESOLUTION

Only developers working on SNMP programming using the SNMP Extendible Agent Management API will need these two .MIB files. To get the two .MIB files:

Download NEWMIB.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download NEWMIB.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get NEWMIB.EXE

The two missing .MIB files will be included with the next release of the Win32 SDK for Windows NT version 3.5.

All licensing and redistribution agreements from the Win32 SDK for Windows NT version 3.5 also apply to these .MIB files. Please consider the two .MIB files part of the Win32 SDK. Please see your licensing agreement for the Win32 SDK for more details.

Additional reference words: 3.50 softlib
KBCategory: kbnetwork kbfile kbprb
KBSubcategory: NtwkSnmp

PRB: WNetEnumResource Returns Different Info on Win95 and NT

PSS ID Number: Q152823

The information in this article applies to:

- Microsoft Win32 Software Development KIT (SDK) for Windows NT, version 3.5
- Microsoft Win32 Software Development KIT (SDK) for Windows 95, version 3.5

SYMPTOMS

When the WINNET example from the January 96 MSDN is built on Windows 95, and the sample is executed at the MS-DOS prompt from within Windows 95, the following error is displayed:

```
   This program has performed an illegal operation and will be shutdown.
   If the problem persists, contact the program vendor.
```

If the WINNET example is executed within the Visual C++ 4.0 debug environment, the following error is received:

```
   Unhandled exception in winnet.exe: 0xC0000005 : Access Violation.
```

CAUSE

The WNetEnumResouce API returns different values on Win95 than on Windows NT.

The problem occurs when an enumeration starts at the root of the network. The first buffer of NETRESOURCE structures returned from WNetEnumResource is different. On Windows 95, the lpProvider field is set to the Network Provider name and the lpRemoteName is set to NULL. On NT, the lpProvider and lpRemoteName fields point to different strings containing the same values.

In the WINNET sample, when the EnumResource function tries to execute its example filter, the _strnicmp C Run-Time function causes an ACCESS VIOLATION because the lpRemoteName is NULL.

RESOLUTION

Checking the value of the lpRemoteName field and ensuring it is pointing to a valid address before allowing the filter to execute prevents the ACCESS VIOLATION error.

STATUS

This behavior is by design.

MORE INFORMATION
================

The difference between the way the WNetEnumResource API works on NT and Windows 95 has only been observed when starting an enumeration at the root of the network. On Windows 95 platforms, it becomes necessary to check the value returned in the lpRemoteName for the first set of structures returned from WNetEnumResource.

Sample Code

```
   /* Compile options needed:
     Be sure to include the mpr.lib library when linking.
   */
   #include <windows.h>
   #include <winnetwk.h>
   #include <stdio.h>

   void main ( void )
   {
      NETRESOURCE resourcebuffer[5000];
      HANDLE hEnum;
      DWORD  netRet, dwEntriesToGet=0XFFFFFFFF;
      DWORD  dwSizeResourcebuffer = sizeof( resourcebuffer);

      //
      // Enumerate starting at the root of the network
      //
      netRet = WNetOpenEnum( RESOURCE_GLOBALNET,
                             RESOURCETYPE_DISK,
                             (RESOURCEUSAGE_CONNECTABLE |
    RESOUCEUSAGE_CONTAINER),
                             (LPNETRESOURCE)NULL,
                             &hEnum);
     if( netRet == NO_ERROR )
     {
        //
        // The WNetOpenEnum was successful. Now get all the network
   providers
    for
        // the network.
        //
        netRet = WNetEnumResource( hEnum,
                                   (LPDWORD)&dwEntriesToGet,
                                   (LPVOID)resourcebuffer,
                                   (LPDWORD)&dwSizeResourcebuffer);
        if( netRet == NO_ERROR )
        {
           //
           // The resourcebuffer contains the network providers. Take a
   look at
           // the lpRemoteName field. ON win 95, this field will be NULL,
   on NT
```

```
            // the field will match the lpProvider field
            //
            DWORD i;
            for( i = 0; i < dwEntriesToGet; i++ )
            {
               if( resourcebuffer[i].lpRemoteName ) printf("Remote Name:
   %s\n",
                     resourcebuffer[i].lpRemoteName);
               else printf("Remote Name: IS NULL\n");
               printf( "Network Provider: %s\n\n",
   resourcebuffer[i].lpProvider );
            }
         }
         else printf("ERROR: WNetEnumResource API failed : %ld\n", netRet );
     }
     else printf("ERROR: WNetOpenEnum API Failed: %ld\n", netRet);

   }
```

Additional reference words: 3.50
KBCategory: kbprg kbnetwork kbprb
KBSubcategory: kbnocat
.END
<><><><><><tegory:
KBSubcategory:
.END

Max Vaughn
SAE, Microsoft Premier Developer Support

PRB: Writing to Resources May Change Last Error

PSS ID Number: Q137289

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SYMPTOMS
========

When writing to an application's resources, the value returned by GetLastError may change. This behavior occurs only in Windows 95, not Windows NT.

CAUSE
=====

When an application writes to its own resources, a page fault occurs, which is handled internally by the kernel. The handling of this page fault may change the value returned by GetLastError.

RESOLUTION
==========

To avoid this problem:

1. Don't write to your own resources. The pointer returned by LockResource should be treated as read-only. This was always the intention, however, writing to your own resources happens to work on Windows NT.

2. If you really need to write to your own resource, do not rely on the last error being preserved. If the value returned by GetLastError is important to you, call GetLastError before writing to your own resource. Then call SetLastError to restore the value after you are finished.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

It is legitimate for an API to change the last error even when it is successful.

Additional reference words: 4.00
KBCategory: kbui kbprb
KBSubcategory: UsrRes

PRB: WSAAsyncSelect() Notifications Stop Coming

PSS ID Number: Q94088

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS
========

I have set a WSAAsyncSelect() call to notify me of read (FD_READ) and disconnection (FD_CLOSE). When a read call is posted on my message queue, I continually read from the socket until there are no more characters waiting. After each read, I use a select() call to determine if more data needs to be read. However, after a while, the notifications stop coming. Why is this?

CAUSE
=====

The message queue must be cleared of extraneous notification messages for each read notification message.

RESOLUTION
==========

Call WSAAsyncSelect(sockt, hWnd, 0, 0) to clear the message queue for each read notification.

MORE INFORMATION
================

Sample Code

```
WSA_READCLOSE:
  if (WSAGETSELECTEVENT( lParam ) == FD_READ) {

    FD_ZERO( &readfds );
    FD_SET( sockt, &readfds);

    timeout.tv_sec = 0;
    timeout.tv_usec = 0;

    /* Clear the queue of any extraneous notification messages. */

    WSAAsyncSelect( sockt, hWnd, 0, 0);

    while (select(0, &readfds, NULL, NULL, &timeout) != 0) {
      recv(sockt, &ch, 1, 0);
      }
```

```
    /* Reset the message notification. */

    WSAAsyncSelect( sockt, hWnd, WSA_READCLOSE, FD_READ | FD_CLOSE);
    }
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork kbprb
KBSubcategory: NtwkWinsock

PRB: WSACancelAsyncRequest Causes a Memory Access Violation

PSS ID Number: Q140166

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0

SYMPTOMS
========

An application that uses the WinSock 1.1 call WSACancelAsyncRequest occasionally throws memory exceptions or causes erratic behavior.

CAUSE
=====

WSACancelAsyncRequest has unavoidable timing conflicts that make canceling calls in progress impossible. WSACancelAsyncRequest is only successful at stopping calls queued for processing that have not yet started.

RESOLUTION
==========

Don't use WSACancelAsyncRequest to cancel calls that may be in progress. Specifically, don't try to cancel asynchronous selects. Allow them to complete, and ignore the results if necessary.

STATUS
======

This behavior is by design.

MORE INFORMATION
================

When an asynchronous WinSock call is in progress, it is hard to stop. For example, an outstanding WSAAsyncSelect call may be in the process of writing to a buffer your application supplied. The sockets DLL has no way of terminating a memory copy because of the timing windows generated by multiple threads. One thread may be in the middle of copying while another does the cancel.

The best way to cancel asynchronous events is to allow them to complete, but ignore the results. For WSAAsyncSelect, you might set a status flag, and when the select message is processed by your window procedure, you can check the status flag, ignoring the message if the flag is set.

Additional reference words: 4.00
KBCategory: kbnetwork kbtshoot kbprb
KBSubcategory: kbntwkmisc

PRB: WSAStartup() May Return WSAVERNOTSUPPORTED on Second Call

PSS ID Number: Q130942

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

SYMPTOMS

If two sections of code within the same process call WSAStartup(), the second call to WSAStartup() fails and returns error WSAVERNOTSUPPORTED unless the second call specifies the version negotiated in the first call.

This happens even if the requested version would normally be accepted. Often the extra calls to WSAStartup() come from one or more DLLs loaded by the process.

RESOLUTION

If multiple calls are made to WSAStartup(), the second call must request the same version negotiated in the first call.

MORE INFORMATION

Some specific examples may help. Currently, if the version of Winsock requested is 1.1 or greater, the negotiated version will be 1.1. If a version less than 1.1 is requested, the call fails and returns the WSAVERNOTSUPPORTED error.

Example One

```
First call : 1.1 requested
Second call: 1.1 requested
Result     : Success
```

Example Two

```
First call : 2.0 requested
Second call: 1.1 requested
Result     : Success
```

Example Three

```
First call : 2.0 requested
```

```
Second call: 2.0 requested
Result     : WSAVERNOTSUPPORTED

Example Four
------------

First call : 1.1 requested
Second call: 2.0 requested
Result     : WSAVERNOTSUPPORTED

Additional reference words: 3.50 4.00 95 3.10
KBCategory: kbnetwork kbprb
KBSubcategory: NtwkWinsock
```

PRB:LVM_HITTEST & ListView State Images Don't Work as Expected

PSS ID Number: Q135786

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SYMPTOMS

The flags member of the LV_HITTESTINFO structure indicates LVHT_NOWHERE, instead of LVHT_ONITEMSTATEICON when you send the LVM_HITTEST message (ListView_HitTest macro) to a ListView control with coordinates that correspond to the state image of an item, and the control is in large or small icon view.

STATUS

This behavior is by design. The ListView state images are not fully implemented in large and small icon view. Future support of state images in large and small icon view has not been decided.

Additional reference words: 4.00 1.30
KBCategory: kbui kbprb
KBSubcategory: UsrCtl

PRB:ReadConsoleInput() & PeekConsoleInput() Don't Receive DBCS

PSS ID Number: Q146448

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows 95

SYMPTOMS
========

The Windows 95 console APIs ReadConsoleInput() and PeekConsoleInput() which are used to read data from a console input buffer do not receive DBCS key codes when the IME is activated.

This behavior is different from Windows NT console support in which the ReadConsoleInput() and PeekConsoleInput() APIs are enabled to receive key event records for converted the DBCS data from the keyboard buffer similar to an IME-enabled Windows-based application. In a Windows NT console application, once the DBCS character is determined, Windows NT posts the converted character bytes to the console buffer.

CAUSE
=====

The Window 95 console architecture is based upon the Windows 3.1 MS-DOS BOX rather than console support in DBCS versions of Windows NT. In Windows 95, the console is working on top of the MS-DOS BOX. This design is opposite to DBCS versions of Windows NT where all console character i/o is directed to the Win32 subsystem.

Specifically, the Windows 95 implementation of ReadConsoleInput() and PeekConsoleInput() hook interrupt 0x09 and return keyboard events back to the application. Because existing MS-DOS IMEs and the VIME VxD hook interrupt 0x16 rather than interrupt 0x09, ReadConsoleInput() and PeekConsoleInput() are unable to receive DBCS characters composed by IMEs.

STATUS
======

This behavior is by design. This architectural difference between DBCS versions of Microsoft Windows 95 and Windows NT is confirmed. The Windows 95 console-IME functionality may change in future versions to become more compatible with Windows NT. We will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: FESDK Far East
KBCategory: kbprg kbprb
KBSubcategory:

PRB:Real-Mode Drivers Don't Support GetFileInformationByHandle

PSS ID Number: Q137234

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SYMPTOMS

GetFileInformationByHandle returns ERROR_NOT_SUPPORTED when called for a file on a volume supported by a real-mode driver.

CAUSE

GetFileInformationByHandle relies on a new service which is not supported by real-mode drivers. The real-mode, handle-based services do not fill in more than a couple of the fields in the output structure.

GetFileInformationByHandle is useful for files that reside on volumes that are supported by protected mode drivers.

RESOLUTION

If you get back ERROR_NOT_SUPPORTED from a GetFileInformationByHandle call, you need to make a name-based call to get the equivalent information.

MS-DOS-based and 16-bit Windows-based applications can call Int 21h function 440Dh, sub function 6Fh (GetDriveMapInfo) to determine if a volume is supported by a protected mode driver.

STATUS

This behavior is by design.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbprb
KBSubcategory: BseFileio

PRB:Scroll Bar Continues Scrolling After Mouse Button Released

PSS ID Number: Q102552

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

The scroll bar continuously scrolls even after the left mouse button is released. The type of scroll bar is irrelevant to this problem, that is, the same problem occurs regardless of whether the scroll bar is part of the window or is a scroll bar control.

CAUSE

This problem occurs usually when a message retrieval loop is executed as the result of actions taken for scrolling upon receiving one of the scroll bar notification messages.

When scrolling, an internal message retrieval loop is started in Windows. The task of this message loop is to keep track of scrolling and to send the appropriate scroll bar notification messages, WM_HSCROLL and WM_VSCROLL. Scrolling is terminated once WM_LBUTTONUP is received. If another message loop is started during scrolling, the WM_LBUTTONUP is retrieved by that message loop, and because an application does not have access to the scroll bar's internal message retrieval loop, WM_LBUTTONUP cannot be dispatched correctly. Therefore, the WM_LBUTTONUP is never received by the internal message retriever, and scrolling is never ended.

The application that is scrolling does not have to retrieve messages explicitly to cause this problem. Calling any of the following functions or processing any message that has a message retrieval loop, while scrolling, can cause the WM_LBUTTONUP to be lost. The functions listed below fall into this category:

```
DialogBox()
DialogBoxIndirect()
DialogBoxIndirectParam()
DialogBoxParam()
GetMessage()
MessageBox()
PeekMessage()
```

RESOLUTION

==========

While Scrolling, the WM_LBUTTONUP message should not be retrieved from the queue by any message retrieval loop other than the scroll bar's internal one.

An application may come across this problem as follows:

- An application implements a message retrieval loop to implement background processing, for example, background processing while performing a time consuming paint.

- An application implements a message retrieval loop to implement communication with another application or DLL. For example, in order to scroll, the application needs to receive data from elsewhere.

Possible Workarounds

Two possible workarounds are listed below. The first workaround is used by many exiting applications and by Windows; however, under rare circumstances the first workaround may not be a feasible one. In this case, the second workaround may be used. However, if possible, please try to avoid implementing message retrieval completely while scrolling.

- Use timer-message-based processing. Break down the complicated processing into smaller tasks and keep track of where each task starts and ends, then perform each task based on a timer message. When all components of the processing are complete, kill the timer. See below for an example of this workaround.

- Implement a message retrieval loop, but make sure WM_LBUTTONUP is not retrieved by it. This can be accomplished by using filters. See below for some examples of this workaround.

Example Demonstrating Workaround 1

An application has a complex paint procedure. Calling ScrollWindow(), to scroll, generates paint messages. Background processing takes place while painting.

1. When receiving the WM_PAINT message do the following:

   a. Call BeginPaint().

   b. Copy the invalidated rect to a global rect variable (for example, grcPaint) to be used in step 2. The global rect grcPaint would be a union of the previously obtained rect (grcPaint) and the new invalidated rect (ps.rcPaint). The code for this will resemble the following:

```
RECT grcPaint;    // Should be initialized before getting the
```

```
                        // first paint message.
            :
            :
        UnionRect(&grcPaint, &ps.rcPaint,&grcPaint);
```

   c. Call ValidateRect() with ps.rcPaint.

   d. Call EndPaint().

   e. Set a Timer.

   This way, no more WM_PAINT messages are generated, because there are no invalid regions, and a timer is set up, which will generate WM_TIMER messages.

2. Upon receiving a WM_TIMER message, check the global rect variable; if it is not empty, take a section and paint it. Then adjust the global rect variable so it no longer includes the painted region.

3. Once the global rect variable is empty then kill the timer.

Example Demonstrating Workaround 2

An application needs to obtain some data through DDE or some other mechanism from another application, which is then displayed in the window. In order to scroll, the application needs to request and then obtain the data from a server application.

There are three different filters that can be used to set up a PeekMessage() and get the information. The filters can be set up by using the uFilterFirst and uFilterLast parameters of PeekMessage(). uFilterFirst specifies the fist message in the range to be checked and uFilterLast specifies the last message in the range to be checked. For more information on PeekMessage() and its parameters, see the Windows SDK "Programmer's Reference, Volume 2: Functions" for version 3.1 and "Reference, Volume 1" for version 3.0.

1. Check and retrieve only the related message(s) for obtaining the needed data.

2. Check for WM_LBUTTONUP without removing it form the queue; if it is in the queue, break. Otherwise, retrieve and dispatch all messages.

3. Retrieve all messages less than WM_LBUTTONUP and greater than WM_LBUTTONUP, but do not retrieve WM_LBUTTONUP.

MORE INFORMATION
==========

Steps to Reproduce Behavior

The following is the sequence of events leading to the loss of the WM_LBUTTONUP message:

1. Click the scroll bar using the mouse.

2. Step 1 generates a WM_NCLBUTTONDOWN message.

3. Step 2 causes a Windows internal message loop to be started. This message loop looks for scroll-bar-related messages. The purpose of this message loop is to generate appropriate WM_HSCROLL or WM_VSCROLL messages. The message loop and scrolling terminates once WM_LBUTTONUP is received.

4. When receiving the WM_HSCROLL or WM_VSCROLL message, the application either gets into a message retrieval loop directly or calls functions which result in retrieval of messages.

5. WM_LBUTTONUP is removed from the queue by the message loop mentioned in step 4. WM_LBUTTONUP is then dispatched.

6. As result of step 5 WM_LBUTTONUP message is dispatched elsewhere and the internal message retrieval loop, mentioned in step 3 never receives it. The message loop in step 3 is looking for the WM_LBUTTONUP to stop scrolling. Because it is not received, the scroll bar continues scrolling.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 scrollbar stuck
KBCategory: kbui kbprb
KBSubcategory: UsrCtl

PRB:Unselecting Edit Control Text at Dialog Box Initialization

PSS ID Number: Q96674

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SYMPTOMS

To remove the highlight (selection) from an edit control text, an EM_SETSEL message must be sent to the control. However, while processing the WM_INITDIALOG message of a dialog box, sending an EM_SETSEL fails to remove the highlight from (unselect) the edit control text.

CAUSE

While processing the WM_INITDIALOG message, sending the EM_SETSEL message fails to remove the highlight from the edit control. This happens because the edit control has not yet been drawn. Because it's not drawn and there is no selection information available to the edit control's procedure, the EM_SETSEL message is ignored. In other words, the SendMessage() function passes the EM_SETSEL message too early to the edit control for it to become effective.

RESOLUTION

There are two solutions to the above problem.

Solution 1

Use SetFocus() to set the input focus on the edit control. Use PostMessage() to post the EM_SETSEL message to the edit control rather than using SendMessage() and return FALSE from the WM_INITDIALOG handler.

Solution 2

When a newly created dialog box is displayed with focus on an edit control, the default text of the edit control is shown highlighted. In some cases, the text highlighting is undesirable because accidentally pressing a character key removes the original text from the edit control. Therefore, the workaround is to unselect the text by sending an EM_SETSEL message to the edit control at the dialog box initialization.

Delay the EM_SETSEL message until the focus is set to the edit control. That is, while processing the first EN_SETFOCUS notification message, an EM_SETSEL message must be sent to the edit control to remove the highlight from its text. For example:

```
   static  BOOL    bFirstTime;   // We want to unselect only once.

   switch ( message )
   {
       case WM_INITDIALOG:
           bFirstTime = TRUE;
           return TRUE;

       case WM_COMMAND:
           switch ( wParam )
           {
               case IDC_EDIT:
                   // If this is the first time, then unselect.
                   if ( HIWORD( lParam ) == EN_SETFOCUS &&
                        bFirstTime )
                   {
                      SendMessage(GetDlgItem( hwndDialog, IDC_EDIT ),
                                  EM_SETSEL, 0,
                      MAKELPARAM( -1, -1 ));
                      bFirstTime = FALSE;
                   }
                   break;
           .
           .
           .
           }  // switch ( wParam )
      .
      .
      .
   } // switch ( message )
```

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbprb
KBSubcategory: UsrCtl

Precautions When Passing Security Attributes

PSS ID Number: Q94839

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

All Win32 APIs that allow security to be specified take a parameter of type LPSECURITY_ATTRIBUTES as the means to attach the security descriptor. However, it is a common error to pass a PSECURITY_DESCRIPTOR type to such functions instead. Because PSECURITY_DESCRIPTOR is of type LPVOID (for opaque data-type reasons), by C Language definition, it is implicitly converted to the correct type. Therefore, the compiler does not generate any warnings; however, unexpected run-time errors will result.

MORE INFORMATION
================

Below is a correct example of creating a named pipe with a security descriptor attached.

Sample Code

```
   saSecurityAttributes.nLength = sizeof(SECURITY_ATTRIBUTES);
   saSecurityAttributes.lpSecurityDescriptor = psdAbsoluteSD;
   saSecurityAttributes.bInheritHandle = FALSE;

   hPipe = CreateNamedPipe(TEST_PIPE_NAME,
                           PIPE_ACCESS_DUPLEX,

   (PIPE_TYPE_BYTE|PIPE_READMODE_BYTE|PIPE_WAIT),
                           100,  // maximum instances
                           0,    // output buffer, sized as needed
                           0,    // input buffer, sized as needed
                           100,  // timeout in milliseconds

   (LPSECURITY_ATTRIBUTES)&saSecurityAttributes);
   if( INVALID_HANDLE_VALUE == hPipe )
   { // handle error
   }
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Preventing Screen Flash During List Box Multiple Update

PSS ID Number: Q66479

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The WM_SETREDRAW message can be used to set and clear the redraw flag for a window. Before an application adds many items to a list box, this message can be used to turn the redraw flag off, which prevents the list box from being painted after each addition. Properly using the WM_SETREDRAW message keeps the list box from flashing after each addition.

MORE INFORMATION

The following four steps outline ways to use the WM_SETREDRAW message to facilitate making a number of changes to the contents of a list box in a visually pleasing manner:

1. Clear the redraw flag by sending the list box a WM_SETREDRAW message with wParam set to FALSE. This prevents the list box from being painted after each change.

2. Send appropriate messages to make any desired changes to the contents of the list box.

3. Set the redraw flag by sending the list box a WM_SETREDRAW message with wParam set to TRUE. The list box does not update its display in response to this message.

4. Call InvalidateRect(), which instructs the list box to update its display. Set the third parameter to TRUE to erase the background in the list box. If this is not done, if a short list box item is drawn over a long item, part of the long item will remain visible.

The following code fragment illustrates the process described above:

```
   /* Step 1: Clear the redraw flag. */
   SendMessage(hWndList, WM_SETREDRAW, FALSE, 0L);

   /* Step 2: Add the strings. */
   for (i = 0; i < n; i++)
      SendMessage(hWndList, LB_ADDSTRING, ...);
```

```
   /* Step 3: Set the redraw flag. */
   SendMessage(hWndList, WM_SETREDRAW, TRUE, 0L);

   /* Step 4: Invalidate the list box window to force repaint. */
   InvalidateRect(hWndList, NULL, TRUE);
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 flash flicker
KBCategory: kbui
KBSubcategory: UsrCtl

Preventing the Console from Disappearing

PSS ID Number: Q99115

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

When a console application is started from the File Manager, from the Program Manager, or by typing "start <progname>" from the command prompt, it executes in its own console. This console disappears as soon as the application terminates, and therefore the user can't read anything written to the screen between the last pause and program exit. To resolve this problem, the programmer should pause the application before termination to allow the user to read all of the information on the screen.

It is not likely that the programmer will want to introduce this pause if the application is started directly from the command prompt, because in this situation it won't make much sense to the user. However, there is no API (application programming interface) that directly determines whether or not the application shares a console with CMD.EXE. There is a method that can be used to determine this information in most cases. When the application first starts up, call GetConsoleScreenBufferInfo(). If the cursor position is (0, 0), then the application has its own console, which will disappear when the application terminates. Otherwise, the application is operating within a console belonging to another program, typically CMD.EXE.

NOTE: This method will not work if the user combines a clear screen (CLS) and execution of the application into one step ([C:\] CLS & <progname>), because the cursor position will be (0, 0), but the application is using the console, which belongs to CMD.EXE.

MORE INFORMATION
================

To start a console application with its own console that will not disappear when the application is terminated, use CMD /K. For example, use

```
start CMD /K <progname>
```

Note that it is possible to programmatically force an application to always have its own console by immediately doing a FreeConsole() and an AllocConsole(). The disadvantage is that the C run-time handles are no longer valid. Use CreateFile("CONIN$", ...) with lpsa->bInherit=TRUE, in combination with _open_osfhandle() and dup2() to close the current handles (stdin, stdout, stderr) and associate handles that will be inherited.

Additional reference words: 3.10 3.50

KBCategory: kbprg
KBSubcategory: BseCon

Preventing Word Wrap in Microsoft Windows Help Files

PSS ID Number: Q88142

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

By default, the Microsoft Windows Help application wraps lines of text to reflect the size of its window. However, there are situations (such as a table of information) in which wrapping text is undesirable. The information below presents two methods of preventing a section of text from changing when the Help application window changes sizes. These techniques apply to version 2.0 of the Microsoft Word for Windows-based application.

MORE INFORMATION

Method 1

This method, which is compatible with versions 3.0 and 3.1 of the Microsoft Windows Help Compiler, involves two steps:

1. Place either a hard or a soft carriage return at the end of each line.

2. Format the section with the "keep lines together" paragraph attribute. From the Format menu, choose Paragraph, and select the Keep Lines Together check box in the Paragraph dialog box.

Method 2

This method, which is compatible only with version 3.1 of the Help Compiler, is to create a one row, one column table in Word for Windows. Set the width of the table as desired and allow the text to wrap within the table normally. Windows Help will duplicate the word breaks in the table provided that the font used to author the table is selected by the Help engine when displaying the table. If Help uses a different font, the text may wrap differently, even though the table keeps the specified width.

Note

When you use either of these methods, if the Windows Help window is not large enough to display the entire width of a topic, Help displays

a horizontal scroll bar rather than wrapping the text to make it visible.

Additional reference words: 3.00 3.10 3.50 4.00 95 HLP word wrap wordwrap engine HC31 HC31.EXE HCP HCP.EXE
KBCategory: kbtool
KBSubcategory: TlsHlp

Primitives Supported by Paths Under Windows 95

PSS ID Number: Q125697

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

Windows 95 supports the full set of Win32 path APIs. However, only a limited set of primitives can be used to build a path. Windows 95 supports the following primitives to build paths:

ExtTextOut
LineTo
MoveToEx
PolyBezier
PolyBezierTo
Polygon
Polyline
PolylineTo
PolyPolygon
PolyPolyline
TextOut

All other Win32 primitives will be ignored if used in a path. As with other Win32 primitives in Windows 95, paths are not mapped pixel-by-pixel to Windows NT; they support only 16-bit coordinates.

Additional reference words: 4.00 GDI
KBCategory: kbgraphic
KBSubcategory: GdiMisc

Printer Escapes Under Windows 95

PSS ID Number: Q125692

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Printer escapes are used to access special printer device features and have been used widely in Windows version 3.x. With Windows 95, Microsoft is encouraging application developers to move away from these escapes by providing GDI functionality to replace them.

For example, a Win32-based application should not call the NEXTBAND and BANDINFO escapes. Banding is no longer needed in Windows 95. Most of these escapes, however, are still provided for 16-bit-based applications for backwards compatibility. The only recommended escapes for 32-bit, Windows 95-based applications are the QUERYESCSUPPORT and PASSTHROUGH escapes.

MORE INFORMATION

================

Applications written for Windows version 3.x can use the QUERYESCSUPPORT and the PASSTHROUGH escapes, as well as the following 10 escapes. It is important to note that these escapes are only supported for backwards compatibility. All new Windows 95-based applications should use Win32 API that replaces these escapes:

```
ABORTDOC
ENDDOC
GETPHYSPAGESIZE
GETPRINTINGOFFSET
GETSCALINGFACTOR
NEWFRAME
NEXTBAND
SETABORTPROC
SETCOPYCOUNT
STARTDOC
```

The following functions should always be used in place of a printer escape:

Function	Printer Escape Replaced
AbortDoc	ABORTDOC
EndDoc	ENDDOC
EndPage	NEWFRAME
SetAbortProc	SETABORTPROC
StartDoc	STARTDOC

Windows 95 provides six new indexes for the GetDeviceCaps function that replace some additional printer escapes:

Index for GetDeviceCaps	Printer Escape Replaced
PHYSICALWIDTH	GETPHYSPAGESIZE
PHYSICALHEIGHT	GETPHYSPAGESIZE
PHYSICALOFFSETX	GETPRINTINGOFFSET
PHYSICALOFFSETY	GETPRINTINGOFFSET
SCALINGFACTORX	GETSCALINGFACTOR
SCALINGFACTORY	GETSCALINGFACTOR

Although a lot of the escapes have been replaced with Win32 GDI equivalent APIs, not all device-dependent escapes have been replaced. It is up to the printer driver manufacturer to decide whether or not its Windows 95-based driver will contain device-specific escapes that were present in its Windows version 3.x driver. An example of a device-specific escape would be the Windows version 3.x PostScript driver's POSTSCRIPT_IGNORE escape. Before calling any of these escapes, an application must first call the QUERESCSUPPORT escape to find out if the escape is supported or not.

Additional reference words: 4.00
KBCategory: kbprint
KBSubcategory: GdiPrn

Printing in Windows Without Form Feeds

PSS ID Number: Q11915

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

A Windows-based application cannot print directly to a printer without issuing a form feed. In 16-bit Windows, the NEWFRAME escape is required to start the print spooler. This escape, in turn, sends a form feed to the printer. Without the NEWFRAME escape, the spooler never runs, and nothing is output to the printer. Win32-based applications should use EndPage(), which replaces the NEWFRAME escape.

MORE INFORMATION

It is possible to drive the spooler directly by using the spooler functions documented in the Windows Device Development Kit (DDK). This allows the printer driver to be bypassed. However, by doing this, the ability to use GDI output functions and Windows's device-independent capabilities will be lost.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbprint
KBSubcategory: GdiPrn

Printing Monochrome and Color Bitmaps from Windows

PSS ID Number: Q64520

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The format of a display bitmap determines the procedure that an application uses to print it. The two display bitmap formats available under Windows are device-dependent bitmaps (DDBs) and device-independent bitmaps (DIBs). DIBs and DIB functions should be used for printing color bitmaps.

MORE INFORMATION
================

An application can use the BitBlt() or StretchBlt() function to print or display a monochrome bitmap. Both printer drivers and display drivers can process monochrome DDBs. However, an application must account for the difference in resolution between a typical display and a typical laser printer. The StretchBlt() function enables an application to appropriately change the size of a monochrome bitmap.

When the display bitmap is a color DDB, printing is more difficult because the display DDB format may not match the printer DDB format. Because Windows supports a wide variety of devices, this situation is quite common. When the formats DDB differ, the application must convert the display DDB into a print DDB or a DIB.

DIBs are designed to ease the process of transferring images between devices. When an application uses a DIB, the GDI or the output driver performs any conversions required for the device. The ShowDIB sample application, provided in the Windows SDK and the Win32 SDK, demonstrates converting a DDB to a DIB and other common manipulations. The file DIB.C is of particular interest. It contains the functions that perform the manipulations. This code can be incorporated into other applications.

For more information, please see the Windows SDK 3.1 DIBView sample or the Win32 SDK WinCap32 sample.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbprint
KBSubcategory: GdiPrn

Printing Offset, Page Size, and Scaling with Win32

PSS ID Number: Q115762

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

The Win32 documentation for the Escape() function says that the GETPHYSPAGESIZE, GETPRINTINGOFFSET, and GETSCALINGFACTOR escapes are obsolete, but it fails to mention the recommended way to get this information.

The information retrieved by all three escapes can now be retrieved by calling GetDeviceCaps() with the appropriate index:

- For the GETPHYSPAGESIZE escape, the indexes to be used with GetDeviceCaps() are PHYSICALWIDTH and PHYSICALHEIGHT.

- For GETPRINTINGOFFSET, the indexes are PHYSICALOFFSETX and PHYSICALOFFSETY.

- For GETSCALINGFACTOR, the indexes are SCALINGFACTORX and SCALINGFACTORY.

All six new indexes are defined in the file WINGDI.H, though they are missing from the GetDeviceCaps() documentation.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprint
KBSubcategory: GdiPrn

Priority Inversion and Windows NT Scheduler

PSS ID Number: Q96418

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

The kernel schedules a thread with the real-time process priority class ahead of every thread with another priority class (nearly all user-mode threads). Windows NT does not alter the priority of real-time threads. The system trusts that the programmer will avoid priority inversion. The remainder of this article talks about the scheduling of threads that are not real-time priority class and how the system solves the problem of priority inversion.

Threads are scheduled according to their priority. When the kernel is choosing which thread will execute on a processor, the highest dynamic (variable) priority thread is picked. Priority inversion occurs when two (or more) threads with different priorities are in contention to be scheduled. Consider a simple case with three threads: Thread 1 is high priority and becomes ready to be scheduled, while thread 2, a low-priority thread, is executing in a critical section. Thread 1, the high-priority thread, begins waiting for a shared resource from thread 2. A third thread has medium priority. The third thread receives all the processor time, because the high-priority thread (thread 1) is busy waiting for shared resources from the low-priority thread (thread 2). Thread 2 won't leave the critical section, because it isn't the highest priority thread and won't be scheduled.

The Windows NT scheduler solves this problem by randomly boosting the priority of threads that are ready to run (in this case the low priority lock-holders). The low priority threads run long enough to let go of their lock (exit the critical section), and the high- priority thread gets the lock back. If the low-priority thread doesn't get enough CPU time to free its lock the first time, it will get another chance on the next scheduling round.

Priority inversion is handled differently in Windows 95. If a high priority thread is dependent on a low priority thread which will not be allowed to run because a medium priority thread is getting all of the CPU time, the system recognizes that the high priority thread is dependent on the low priority thread and will boost the low priority thread's priority up to the priority of the high priority thread. This will allow the formerly low priority thread to run and unblock the high priority thread that was waiting on it.

MORE INFORMATION
================

Each Process has a base priority. Each thread has a base priority that is a function of its process base priority. A thread's base priority is settable to:

- 1 or 2 points above the process base
- equal to the process base
- 1 or 2 points below the process base

Priority setting is exposed through the Win32 API. In addition to a base priority, all threads have a dynamic priority. The dynamic priority is never less than the base priority. The system raises and lowers the dynamic priority of a thread as needed.

All scheduling is done strictly by priority. The scheduler chooses the highest priority thread which is ready to run. On a multi-processor (MP) system, the highest N runnable threads run (where N is the number of processors). The thread priority used to make these decisions is the dynamic priority of the thread.

When a thread is scheduled, it is given a quantum of time in which to run. The quantum is in units of clock ticks. The system currently uses 2 units of quantum (10ms on r4000 and 15ms on x86).

When a thread is caught running during the clock interrupt, its quantum is decremented by one. If the quantum goes to zero and the thread's dynamic priority is not at the base priority, the thread's dynamic priority is decremented by one and the thread's quantum is replenished. If a priority change occurs, then the scheduler locates the highest priority thread which is ready to run. Otherwise, the thread is placed at the end of the run queue for it's priority allowing threads of equal priority to be "round robin" scheduled. The above is a description of what is usually called priority decay, or quantum and priority decay.

When a thread voluntarily waits (an an event, for I/O, etc), the system will usually raise the thread's dynamic priority when it resumes. Internally, each wait type has an associated priority boost. For example, a wait associated with disk I/O has a one point dynamic boost. A wait associated with a keyboard I/O has a 5 point dynamic boost. In most cases, this boost will raise the priority of the thread such that it can be scheduled very soon afterwards, if not immediately.

There are other circumstances under which priority will be raised. For example, whenever a window receives input (timer messages, mouse move messages, etc), an appropriate boost is given to all threads within the process that owns the window. This is the boost that allows a thread to reshape the mouse pointer when the mouse moves over a window.

By default, the foreground application has a base process priority that is one point higher than the background application. This allows the foreground process to be even more responsive. This can be changed by bringing up the System applet, selecting the Tasking button, and choosing a different option.

Additional reference words: 3.10 3.50

KBCategory: kbprg
KBSubcategory: BseProcThrd

Process Will Not Terminate Unless System Is In User-mode

PSS ID Number: Q92761

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

Under Windows NT, a process will not be terminated unless the system is in user-mode. Suppose that TerminateProcess() is called while a device driver or filesystem code is being executed. The system will wait until the threads are running user code before marking the process for termination. On system exit, processes that were the target of a TerminateProcess() will be killed.

This may affect drivers. If a driver is waiting for an object or multiple objects in WaitMode or UserMode, its wait may complete unsuccessfully due to a termination request. Any code that does a UserMode wait or an Alertable wait must check the return status of the wait call. If the wait fails with STATUS_USER_APC or STATUS_ALERTED, this is not an error. The driver should cleanup and return to user-mode.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseProcThrd

Process WM_GETMINMAXINFO to Constrain Window Size

PSS ID Number: Q67166

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Microsoft Windows sends a WM_GETMINMAXINFO message to a window to determine the maximized size or position for the window, and the maximum or minimum tracking size for the window. An application can change these parameters by processing the WM_GETMINMAXINFO message.

Each window type has an absolute minimum size. If an application changes any of the values associated with WM_GETMINMAXINFO to a value smaller than the minimum, Windows will override the values specified by the application and use the minimum size. This minimum window size restriction has been removed from Windows version 3.1.

Note that Windows can send a WM_GETMINMAXINFO message to a window prior to sending a WM_CREATE message. Therefore, any processing for the WM_GETMINMAXINFO message must be independent of processing done for the WM_CREATE message.

MORE INFORMATION

An application can use the WM_GETMINMAXINFO message to constrain the size of a window. For example, the application can prevent the user from changing a window's width while allowing the user to affect its height, or vice versa. The following code demonstrates fixing the width:

```
  int width;
  LPPOINT lppt;
  RECT rect;

  case WM_GETMINMAXINFO:
      lppt = (LPPOINT)lParam;   // lParam points to array of POINTs

      GetWindowRect(hWnd, &rect);   // Get current window size
      width = rect.right - rect.left + 1;

      lppt[3].x = width   // Set minimum width to current width
      lppt[4].x = width   // Set maximum width to current width
```

```
      return DefWindowProc(hWnd, message, wParam, lParam);
```

The modifications required to fix the height are quite straightforward.

For more information on the array of POINT structures that accompanies the WM_GETMINMAXINFO message, please refer to the "Microsoft Windows Software Development Kit Reference."

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Processes Maintain Only One Current Directory

PSS ID Number: Q84244

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

Processes under Windows NT maintain only one current directory. Under MS-DOS or OS/2, a process will maintain a current directory for each drive.

MORE INFORMATION
================

For example, if you do the following

1. Set the current drive to be drive C and set the current directory to be \MAINC\MAINSUBC.

2. Change the current drive to be drive D and set the current directory to be \MAIND\MAINSUBD.

when you reset the current drive to drive C, the current directory will be the original directory: \MAINC\MAINSUBC.

MS-DOS and OS/2 use a current directory structure (CDS) to maintain this information. The memory for this structure is allocated at boot time, and is set by the LASTDRIVE= line in the CONFIG.SYS file. For example, if you set LASTDRIVE=Z, you will have 26 entries in the CDS and will be able to track 26 current directories.

Windows NT by default allows a process to track only one current directory--the one for the current drive--because the underlying operating system does not use drive letters; it always uses fully-qualified names such as:

```
\Device\HardDisk0\Partition1\autoexec.bat
```

The Win32 subsystem maintains drive letters by setting up symbolic links such as:

```
\DosDevices\C: == \Device\HardDisk0\Paritition1
\DosDevices\D: == \Device\HardDisk0\Paritition2
\DosDevices\E: == \Device\HardDisk1\Paritition1
```

(Partitions are 1-based while hard disks are 0-based because Partition0 refers to the entire physical device, which is the "file" that FDISK opens to do its work.) Therefore, when you do SetCurrentDirectory("c:\tmp\sub"), the Win32 subsystem translates that to "\DosDevices\c:\tmp\sub", "...".

As far as Windows NT is concerned, there are no "drives," there is one object namespace.

CMD.EXE maintains a private current directory for each drive it has touched and uses environment variables to associate a current directory with each drive. These environment variables have the form "=<drive>:".

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

Processing CBN_SELCHANGE Notification Message

PSS ID Number: Q66365

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

When a combo box receives a CBN_SELCHANGE notification message, GetDlgItemText() will give the text of the previous selection and not the text of the new selection.

To get the text of the new selection, send the CB_GETCURSEL message to retrieve the index of the new selection and then send a CB_GETLBTEXT message to obtain the text of that item.

MORE INFORMATION

================

When an application receives the CBN_SELCHANGE notification message, the edit/static portion of the combo box has not been updated. To obtain the new selection, send a CB_GETLBTEXT message to the combo box control. This message places the text of the new selection in a specified buffer. The following is a brief code fragment:

```
   ...  /* other code */

   case CBN_SELCHANGE:
      hCombo = LOWORD(lParam);  /* Get combo box window handle */

    /* Get index of current selection and then the text of that selection
*/

      index = SendMessage(hCombo, CB_GETCURSEL, (WORD)0, 0L);
      SendMessage(hCombo, CB_GETLBTEXT, (WORD)index, (LONG)buffer);
      break;

   ...  /* other code */
```

NOTE: For Win32 applications, change the WORD and LONG casts to WPARAM and LPARAM, respectively.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 combobox

KBCategory: kbui

KBSubcategory: UsrCtl

Processing WM_PALETTECHANGED and WM_QUERYNEWPALETTE

PSS ID Number: Q77702

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

An application that manipulates the system palette should process the WM_PALETTECHANGED and WM_QUERYNEWPALETTE messages to maintain its appearance during system palette and input focus changes.

MORE INFORMATION
================

The WM_PALETTECHANGED message informs all windows that the window with input focus has realized its logical palette, thereby changing the system palette. This message allows a window without input focus that uses a color palette to realize its logical palettes and update its client area.

This message is sent to all windows, including the one that changed the system palette and caused this message to be sent. The wParam of this message contains the handle of the window that caused the system palette to change. To avoid an infinite loop, care must be taken to check that the wParam of this message does not match the window's handle. The following sample code demonstrates how to process WM_PALETTECHANGED:

```
    case WM_PALETTECHANGED:
        {
        HDC hDC;           // Handle to device context
        HPALETTE hOldPal;  // Handle to previous logical palette

        // If this application did not change the palette, select
        // and realize this application's palette
        if (wParam != hWnd)
           {
           // Need the window's DC for SelectPalette/RealizePalette
           hDC = GetDC(hWnd);

           // Select and realize hPalette
           hOldPal = SelectPalette(hDC, hPalette, FALSE);
           RealizePalette(hDC);
```

```
            // When updating the colors for an inactive window,
            // UpdateColors can be called because it is faster than
            // redrawing the client area (even though the results are
            // not as good)
            UpdateColors(hDC);

            // Clean up
            if (hOldPal)
               SelectPalette(hDC, hOldPal, FALSE);
            ReleaseDC(hWnd, hDC);
            }
         }
         break;
```

NOTE: The WM_PALETTECHANGED message is sent to all top-level and overlapped windows; therefore, if any child window uses a color palette, this message must be passed on to it.

The WM_QUERYNEWPALETTE message informs a window that it is about to receive input focus. In response, the window receiving focus should realize its palette as a foreground palette and update its client area. If the window realizes its palette, it should return TRUE; otherwise, it should return FALSE. The following sample code demonstrates processing WM_QUERYNEWPALETTE:

```
    case WM_QUERYNEWPALETTE:
        {
        HDC hDC;           // Handle to device context
        HPALETTE hOldPal;  // Handle to previous logical palette

        // Need the window's DC for SelectPalette/RealizePalette
        hDC = GetDC(hWnd);

        // Select and realize hPalette
        hOldPal = SelectPalette(hDC, hPalette, FALSE);
        RealizePalette(hDC);

        // Redraw the entire client area
        InvalidateRect(hWnd, NULL, TRUE);
        UpdateWindow(hWnd);

        // Clean up
        if (hOldPal)
           SelectPalette(hDC, hOldPal, FALSE);
        ReleaseDC(hWnd, hDC);

        // Message processed, return TRUE
        return TRUE;
        }
```

NOTE: The WM_QUERYNEWPALETTE message is sent to all top-level and overlapped windows; therefore, if any child window uses a color palette, this message must be passed on to it.

Additional reference words: 3.00 3.10 3.50 4.00 95

KBCategory: kbgraphic
KBSubcategory: GdiPal

Programatically Appending Text to an Edit Control

PSS ID Number: Q109550

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Windows-based applications often use edit controls to display text. These applications sometimes need to append text to the end of an edit control instead of replacing the existing text. There are several ways to do this in Windows:

- Use the EM_SETEL and EM_REPLACESEL messages.

  -or-

- Use the EM_SETSEL message with the clipboard functions to append text to the edit control's buffer.

MORE INFORMATION
================

The EM_SETSEL message can be used to place a selected range of text in a Windows edit control. If the starting and ending positions of the range are set to the same position, no selection is made and a caret can be placed at that position. To place a caret at the end of the text in a Windows edit control and set the focus to the edit control, do the following:

```
HWND hEdit = GetDlgItem (hDlg, ID_EDIT);
int ndx = GetWindowTextLength (hEdit);
SetFocus (hEdit);
SendMessage (hEdit, EM_SETSEL, 0, MAKELONG (ndx, ndx));
```

Once the caret is placed at end in the edit control, the EM_REPLACESEL message can be use to append text to the edit control. An application sends an EM_REPLACESEL message to replace the current selection in an edit control with the text specified by the lpszReplace (lParam) parameter. Because there is no current selection, the replacement text is inserted at the current cursor location. This example sets the selection to the end of the edit control and inserts the text in the buffer:

```
SendMessage (hEdit, EM_SETSEL,     0, MAKELONG (ndx, ndx));
SendMessage (hEdit, EM_REPLACESEL, 0, (LPARAM) ((LPSTR) szBuffer));
```

One other way of inserting text into an edit control is to use the Windows

clipboard. If the application has the clipboard open or finds it convenient to open the clipboard, and copies the text into the clipboard, then it can send the WM_PASTE message to the edit control to append text.

Before sending the WM_PASTE message, the caret must be placed at the end of the edit control text using the EM_SETSEL message. Below is pseudo code that shows how to implement this method:

```
   OpenClipBoard () ;
   EmptyClipBoard() ;
   SetClipBoardData() ;

   SendMessage (hEdit, EM_SETSEL,  0, MAKELONG (ndx, ndx));
   SendMessage (hEdit, WM_PASTE,   0, 0L);
```

This pseudo code appends text to the end of the edit control. Note that the data in the clipboard must be in CF_TEXT format.

NOTE: For Win16 programs, the EM_SETSEL message specifies that the LPARAM LOWORD/HIWORD control the start/end. For Win32 programs, the EM_SETSEL message specifies that the WPARAM/LPARAM control the selection start/end.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Programatically Using ICMP Echo Request and Reply (Ping)

PSS ID Number: Q139459

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, and 4.0

It is possible to write applications that programatically use the Internet Control Message Protocol (ICMP) echo request and reply (popularly known as "ping"). For more details, please look in the ICMP directory of the Win32 SDK compact disc.

Please note that this mechanism is temporary, so there is no section in the Win32 Help file that addresses it. This will be replaced in the future by more powerful mechanisms supported by Winsock 2.0.

REFERENCES

==========

Win32 Compact Disc \Mstools\Icmp.

Additional reference words: 3.50 3.51 4.00 Windows 95

KBCategory: kbnetwork

KBSubcategory:

Propagating Environment Variables to the System

PSS ID Number: Q104011

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

User environment variables can be modified using the System control panel application or by editing the following Registry key:

```
HKEY_CURRENT_USER \
      Environment
```

System environment variables can be modified using the System control panel application (in Windows NT 3.5 and later) or by editing the following Registry key:

```
HKEY_LOCAL_MACHINE \
            SYSTEM \
 CurrentControlSet \
           Control \
   Session Manager \
       Environment
```

Note, however, that modifications to the environment variables do not result in immediate change. For example, if you start another Command Prompt after making the changes, the environment variables will reflect the previous (not the current) values. The changes do not take effect until you log off and then log back on.

To effect these changes without having to log off, broadcast a WM_WININICHANGE message to all windows in the system, so that any interested applications (such as Program Manager, Task Manager, Control Panel, and so forth) can perform an update.

MORE INFORMATION
================

For example, on Windows NT, the following code fragment should propagate the changes to the environment variables used in the Command Prompt:

```
SendMessage( FindWindow( "Progman", NULL ), WM_WININICHANGE,
  0L, (LPARAM) "Environment" );
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrMisc

Prototypes for SetSystemCursor() & LoadCursorFromFile()

PSS ID Number: Q122564

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.5

SUMMARY

=======

The function prototypes for SetSystemCursor() and LoadCursorFromFile() were inadvertently omitted from the Win32 SDK header files. These APIs are resolved by linking for USER32.LIB.

Additionally, the use of the OCR_* constants as described in the online help for LoadCursorFromFile() is not currently implemented. However, this functionality is available through LoadCursor().

MORE INFORMATION

================

The correct function prototypes are given below. NOTE: These prototypes were included correctly in the Win32 SDK 3.51/4.0 documentation.

To use these functions, add the prototypes to a file in your project after including WINDOWS.H.

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/* SetSystemCursor prototype */
WINUSERAPI BOOL WINAPI SetSystemCursor (HCURSOR hcur, DWORD id);

/* LoadCursorFromFile prototypes - UNICODE aware */
WINUSERAPI HCURSOR WINAPI LoadCursorFromFileA (LPCSTR lpFileName);
WINUSERAPI HCURSOR WINAPI LoadCursorFromFileW (LPCWSTR lpFileName);

#ifdef UNICODE
#define LoadCursorFromFile  LoadCursorFromFileW
#else
#define LoadCursorFromFile  LoadCursorFromFileA
#endif // !UNICODE

#ifdef __cplusplus
}
#endif  /* __cplusplus */
```

Additional reference words: 3.50

KBCategory: kbgraphic kbdocerr

KBSubcategory: GdiCurico

Providing a Custom Wordbreak Function in Edit Controls

PSS ID Number: Q109551

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

An application sends the EM_SETWORDBREAKPROC message to an edit control to replace the default wordwrap function with an application-defined wordwrap function. The default wordwrap function breaks a line in a multiline edit control (MLE) at a space character. If an application needs to change this functionality (that is, to break at some character other than a space), then the application must provide its own wordwrap (wordbreak) function.

MORE INFORMATION

A wordwrap function scans a text buffer (which contains text to be sent to the display), looking for the first word that does not fit on the current display line. The wordwrap function places this word at the beginning of the next line on the display.

A wordwrap function defines the point at which Windows should break a line of text for multiline edit controls, usually at a space character that separates two words. This can be changed so that the line in an MLE can be broken at any character. For more information on the EM_SETWORDBREAKPROC message, please refer to the Windows 3.1 SDK "Programmer's Reference, Volume 3: Messages, Structures, and Macros" manual.

Below is sample code that demonstrates how to break a line in a multiline edit control at the "~" (tilde) character (for example) instead of the regular space (" ") character.

The sample code assumes that the edit control is a multiline edit control and that it is a child control in a dialog box.

Sample Code

```
//Prototype the application-defined wordbreakproc.
int CALLBACK WordBreakProc(LPSTR, int, int, int) ;

//Install wordbreakproc in the WM_INITDIALOG case.
  case WM_INITDIALOG:
```

```
    lpWrdBrkProc = MakeProcInstance(WordBreakProc, hInst);


    //Send the EM_SETWORDBREAKPROC message to the edit control
    //to install the new wordbreak procedure.
     SendDlgItemMessage(hDlg, ID_EDIT, EM_SETWORDBREAKPROC, 0,
                        (LPARAM)(EDITWORDBREAKPROC)lpWrdBrkProc) ;
     return (TRUE);

int FAR PASCAL WordBreakProc(LPSTR lpszEditText, int ichCurrent,
                             int cchEditText, int wActionCode)
{

   char FAR *lpCurrentChar;
   int  nIndex;
   int  nLastAction;

   switch (wActionCode) {

     case WB_ISDELIMITER:

     // Windows sends this code so that the wordbreak function can
     // check to see if the current character is the delimiter.
     // If so, return TRUE. This will cause a line break at the ~
     // character.

      if ( lpszEditText[ichCurrent] == '~' )
         return TRUE;
      else
         return FALSE;

      break;

     // Because we have replaced the default wordbreak procedure, our
     // wordbreak procedure must provide the other standard features in
     // edit controls.

     case WB_LEFT:

     // Windows sends this code when the user enters CTRL+LEFT ARROW.
     // The wordbreak function should scan the text buffer for the
     // beginning of the word from the current position and move the
     // caret to the beginning of the word.

         {
            BOOL bCharFound = FALSE;

            lpCurrentChar = lpszEditText + ichCurrent;
            nIndex = ichCurrent;

            while (nIndex > 0  &&
                  (*(lpCurrentChar-1) != '~' &&
                   *(lpCurrentChar-1) != 0x0A) ||
                  !bCharFound )
            {
```

```
                lpCurrentChar = AnsiPrev(lpszEditText ,lpCurrentChar);
                nIndex--;

                if (*(lpCurrentChar) != '~' &&  *(lpCurrentChar) != 0x0A )

                   // We have found the last char in the word. Continue
                   // looking backwards till we find the first char of
                   // the word.
                  {
                    bCharFound = TRUE;

                    // We will consider a CR the start of a word.
                    if (*(lpCurrentChar) == 0x0D)
                        break;
                  }

              }
              return nIndex;

           }
       break;

      case WB_RIGHT:

      //Windows sends this code when the user enters CTRL+RIGHT ARROW.
      //The wordbreak function should scan the text buffer for the
      //beginning of the word from the current position and move the
      //caret to the end of the word.

       for (lpCurrentChar = lpszEditText+ichCurrent, nIndex = ichCurrent;
            nIndex < cchEditText;
            nIndex++, lpCurrentChar=AnsiNext(lpCurrentChar))

        if ( *lpCurrentChar == '~' ) {
       lpCurrentChar=AnsiNext(lpCurrentChar);
          nIndex++;

          while ( *lpCurrentChar == '~' ) {
            lpCurrentChar=AnsiNext(lpCurrentChar);
            nIndex++;
           }

          return nIndex;
        }

        return cchEditText;
        break;

   }
}
```

The wordwrap (wordbreak) function above needs to be exported in the .DEF file of the application. The function can be modified and customized according to the application's needs.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 multi-line
KBCategory: kbui
KBSubcategory: UsrCtl

PSTR's in OUTLINETEXTMETRIC Structure

PSS ID Number: Q90085

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

The OUTLINETEXTMETRIC structure ends with four fields of type PSTR. The four fields in question are not actually absolute pointers. They are offsets from the beginning of the OUTLINETEXTMETRIC structure to the strings in question, as the documentation indicates:

otmpFamilyName
Specifies the offset from the beginning of the structure to a string specifying the family name for the font.

otmpFaceName
Specifies the offset from the beginning of the structure to a string specifying the face name for the font. (This face name corresponds to the name specified in the LOGFONT structure.)

otmpStyleName
Specifies the offset from the beginning of the structure to a string specifying the style name for the font.

otmpFullName
Specifies the offset from the beginning of the structure to a string specifying the full name for the font. This name is unique for the font and often contains a version number or other identifying information.

The only difference between this structure in Windows 3.1 and Windows NT is that the strings may be stored in either Unicode or ASCII under Windows NT.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbgraphic
KBSubcategory: GdiMisc

Querying and Modifying the States of System Menu Items

PSS ID Number: Q83453

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

An application should query or set states of the Restore, Move, Size, Minimize, and Maximize items on the system menu during the processing of a WM_INITMENU or a WM_INITMENUPOPUP message.

MORE INFORMATION
================

Windows changes the state of the Restore, Move, Size, Minimize, and Maximize items on the system menu just before it draws the menu on the screen and sends the WM_INITMENU and WM_INITMENUPOPUP messages.

Windows sets the states of these menu items according to the state of the window just before the menu is displayed. For example, if the window is minimized when its system menu is pulled down, the Minimize menu item is unavailable (grayed). If an overlapped window is maximized when its system menu is pulled down, the Move, Size, and Maximize items are unavailable.

If an application queries or sets the state of any of these system menu items, the query or change should occur during the processing of the WM_INITMENU or WM_INITMENUPOPUP message. If any menu item state is queried before one of these messages is processed, it could reflect a previous state of the window. If any state is set before one of these messages is processed, Windows will reset the menu items to correspond to the state of the window just prior to sending these messages.

Windows does not change the state of the Close menu item. Its state can be changed or queried at any time.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMen

Querying Device Support for MaskBlt

PSS ID Number: Q108929

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

The Win32 documentation for MaskBlt() states:

> Not all devices support the MaskBlt function. An application should call the GetDeviceCaps function to determine whether a device supports this function.

To query support for MaskBlt(), an application should query the device for BitBlt support by passing RC_BITBLT constant to GetDeviceCaps().

MaskBlt() implements transparent blts in Windows NT. For more information, please see the following article in the Microsoft Knowledge Base:

> ARTICLE-ID: Q89375
> TITLE : Transparent Blts in Windows NT

GDI implements this application programming interface (API) by calling BitBlt(). Because BitBlt() is implemented at the driver level, applications that calls MaskBlt() should check for BitBlt() support on the device.

Additional reference words: 3.10 3.50
KBCategory: kbgraphic
KBSubcategory: GdiBmp

RAS Server Does Not Support the Full NetBIOS 3.0 Specification

PSS ID Number: Q149685

The information in this article applies to:

- Microsoft Windows NT Advanced Server versions 3.5, 3.51, 4.0
- Microsoft Windows NT Workstation versions 3.5, 3.51, 4.0
- Microsoft Win32 Software Development Kit versions 3.5, 3.51, 4.0

SUMMARY

Windows NT Remote Access Server (RAS) supports most NetBIOS 3.0 commands with a few exceptions. Three features are not supported:

- Sessions can only be established with a unique name, not a group name.

- The NCBFINDNAME command is unsupported.

- The first byte of a NetBIOS name must be a non-zero value.

MORE INFORMATION

The NetBIOS support in RAS was designed to support Windows NT redirector over NetBIOS, not to be fully compliant with the NetBIOS specification. A secondary goal was to support the majority of existing NetBIOS applications (like Lotus Notes).

All widely used NetBIOS commands are supported, and the few that are not supported are both uncommon and complex to implement in a RAS context.

REFERENCES

IBM NetBIOS 3.0 Specification

Additional reference words: 3.50 3.51 4.00 netbios group unique findname ras rasapi win32
KBCategory: kbprg kbtshoot
KBSubcategory:

Raster and Stroke Fonts; GDI and Device Fonts

PSS ID Number: Q77126

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

NOTE: The information contained in this article does not address TrueType fonts. For information on TrueType fonts, please see chapter 18 of "Guide to Programming" for the Windows SDK version 3.1.

In Windows version 3.0, there are two different ways that the graphical device interface (GDI) can generate characters for a font. For a raster font, GDI displays the font by copying bitmaps to the output device. For a stroke font, GDI displays the font by drawing lines between a series of points that describe each character. Each font is owned by either GDI or by a specific device. Type and ownership information can be determined by enumerating the fonts. This article discusses the two font types and two font ownership types.

MORE INFORMATION
================

A raster font stores its characters as a series of bitmaps; a stroke font stores its characters as a set of vector operations that describe the characters. When a character from a raster font is drawn, the bitmap is copied onto the device. When a character from a stroke font is displayed, the lines are drawn connecting the points that describe the character. Examples of raster fonts provided with Windows are Courier and Helv; examples of stroke fonts are Script and Roman.

Raster fonts are only available in specific sizes. Some devices can scale installed raster fonts to integer multiples of their size. Use the GetDeviceCaps() function to determine whether the device has this capability. The Windows GDI will scale its raster fonts as required regardless of the device capability. Stroke fonts can be scaled to any size and can also be rotated.

GDI fonts are owned by the GDI; they are available to all devices. Device fonts are fonts that are owned by a particular device; they are available only on that device.

By enumerating the fonts, an application can determine which ones are raster or stroke fonts, and which are GDI or device fonts. The callback

function used with EnumFonts() has the parameter nFontType. As stated on page 4-118 of the "Microsoft Windows Software Development Kit Reference Volume 1," the bitwise AND (&) operator can be used with the constants RASTER_FONTTYPE and DEVICE_FONTTYPE to determine the font type. If the RASTER_FONTTYPE bit is set, the font is a raster font; otherwise, it is a stroke font. If the DEVICE_FONTTYPE bit is set, the font is owned by the device that corresponds to the display context handle (HDC) used in the EnumFont() call; otherwise it is a GDI font.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiFnt

RCDATA Begins on 32-Bit Boundary in Win32

PSS ID Number: Q84081

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

RCDATA is guaranteed to begin on a DWORD boundary. However, the strings and the integers specified in the statement are not aligned by the Resource Compiler (RC)

RCDATA statement:

```
resname RCDATA
BEGIN

    0,0,

END
```

The definition of RCDATA is not changed. The strings and integers specified in the statement, 0 in this case, are not aligned on the DWORD boundary. However, the beginning of the data is DWORD-aligned.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsRc

Reasons for Failure of Clipboard Functions

PSS ID Number: Q92530

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The following clipboard functions:

   OpenClipboard()
   CloseClipboard()
   EmptyClipboard()
   GetClipboardData()
   SetClipboardData()
   EnumClipboardFormats()
   SetClipboardViewer()
   ChangeClipboardChain()
   GetOpenClipboardWindow()
   GetClipboardOwner()

can fail for several reasons. Different functions return different values to indicate failure. Read the documentation for information about each function. This article combines the causes of failure for all functions and provides a resolution or explanation. In the More Information section, a list of affected functions follows each cause. The causes are:

1. The clipboard is not opened by any application.

2. The current application does not have the clipboard open.

3. The current application does not own the clipboard.

4. User's data segment is full.

5. Insufficient global memory.

6. The specified clipboard format is not supported.

7. The application that set the clipboard data placed a corrupt or invalid metafile in the clipboard.

8. An application is attempting to open an already open clipboard. The debug mode of Windows 3.1 will send the "Clipboard already open" message.

9. The application that opened the clipboard used NULL as the window
   handle.

MORE INFORMATION
================

Cause 1: The clipboard is not opened by any application.

Resolution 1: Open the clipboard using OpenClipboard(). If a DLL needs
to open the clipboard, it may pass hwnd = NULL to OpenClipboard().

Explanation 1: An application cannot copy data (using
SetClipboardData()) when no application has the clipboard currently
open.

Affected Functions: SetClipboardData().

Cause 2: The current application does not have the clipboard open.

Resolution 2: Open the clipboard using OpenClipboard(). If a DLL needs
to open the clipboard, it may pass hwnd = NULL to OpenClipboard().

Explanation 2: An application cannot empty or close the clipboard without
first opening it.

Affected Functions: EmptyClipboard(), CloseClipboard().

Cause 3: The current application does not own the clipboard.

Resolution 3: Open the clipboard and get ownership by emptying it.

Explanation 3: An application cannot enumerate the clipboard formats
without owning it.

Affected Functions: EnumClipboardFormats().

Cause 4: User's data segment is full.

Explanation 4: There should be space available in User's data segment to
store internal data structures when SetClipboardData() is called.

Affected Function: SetClipboardData().

Cause 5: Insufficient global memory.

Explanation 5: If the clipboard has data in either the CF_TEXT or
CF_OEMTEXT format and if GetClipboardData() requests text in the
unavailable format, then Windows will perform the conversion. The
converted text must be stored in global memory.

Affected Function: GetClipboardData().

Cause 6: The specified clipboard format is not supported.

Resolution 6: Use IsClipboardFormatAvailable() to check whether the specified format is available on the clipboard.

Affected Function: GetClipboardData().

Cause 7: The application that set the clipboard data placed a corrupt or invalid metafile in the clipboard.

Resolution 7: There are no functions to tell whether a given metafile is corrupt or invalid. Try playing the metafile and see if the metafile plays as expected.

Affected Function: SetClipboardData().

Cause 8: Application is attempting to open an already open clipboard. The debug mode of Windows 3.1 will send the "Clipboard already open" message.

Explanation 8: The clipboard must be closed by the application that opened it, before other applications can open it.

Affected Functions: OpenClipboard().

Cause 9: The application that opened the clipboard used NULL as the window handle.

Explanation 9: An application can call OpenClipboard(NULL) to successfully open a clipboard. The side effects are that subsequent calls to GetClipboardOwner() and GetOpenClipboardWindow() return NULL. An application can also call SetClipboardViewer(NULL) successfully. However, there is no reason why this should be allowed, and it is currently reported as a bug. The side effects are that subsequent calls to GetClipboardViewer() and ChangeClipboardChain() return NULL. NULL from these functions does not necessarily imply that they failed.

Affected Functions: GetClipboardOwner(), GetOpenClipboardWindow(), GetClipboardViewer(), ChangeClipboardChain().

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
GetClipboardFormatName
RegisterClipboardFormat
KBCategory: kbui
KBSubcategory: UsrClp

Reasons for Failure of Menu Functions

PSS ID Number: Q89739

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The Menu functions (AppendMenu(), CheckMenuItem(), CreateMenu(), CreatePopupMenu(), DeleteMenu(), DestroyMenu(), GetMenu(), GetMenuItemID(), GetMenuString(), GetSubMenu(), GetSystemMenu(), HiliteMenuItem(), InsertMenu(), LoadMenuIndirect(), ModifyMenu(), RemoveMenu(), SetMenu(), SetMenuItemBitmaps(), and TrackPopupMenu()) can fail for several reasons. Different functions return different values to indicate failure. Read the documentation for information about each function. This article combines the causes of failure for all functions and provides a resolution or explanation. A list of affected functions follows each cause. The causes are:

1. Invalid hWnd parameter.

2. Invalid hMenu parameter.

3. The menu item is not found.

4. No space left in User's heap to hold a string or to hold an internal data structure for owner draw menu items or to create a menu or to create a window for TrackPopupMenu().

5. There are no items in the menu.

6. The menu resource could not be found (FindResource()) or loaded (LoadResource()) or locked (LockResource()) in memory.

7. TrackPopupMenu() is called while another popup menu is being tracked in the system.

8. The hMenu that has been passed to TrackPopupMenu() has been deleted.

9. MENUITEMTEMPLATEHEADER's versionNumber field is non-zero.

MORE INFORMATION

Cause 1: Invalid hWnd parameter.

Resolution 1: Validate the hWnd parameter using IsWindow(). Make sure that hWnd is not a child window.

NOTE: Resolution 1 does not apply to TrackPoupuMenu().

Explanation 1: In Windows, menus are always associated with a window. Child windows cannot have menu bars.

Affected Functions: All functions that take hWnd as a parameter except for TrackPoupuMenu().

Cause 2: Invalid hMenu parameter.

Resolution 2: Validate hMenu with IsMenu().

Affected Functions: All functions that take hMenu as a parameter.

Cause 3: The menu item is not found.

Resolution 3: If the menu item is referred to BY_POSITION, make sure that the index is lesser than the number of items. If the menu item is referred to BY_COMMAND, an application has to devise its own method of validating it.

Explanation 3: Menu items are numbered consecutively starting from 0. Remember that separator items are also counted.

Affected Functions: All functions that refer to a menu item.

Cause 4: No space left in User's heap to hold a string or to hold an internal data structure for owner draw menu items or to create a menu.

Resolution 4: Remember to delete all menus and other objects that have been created by the application when they are not needed any more. If you suspect that objects left undeleted by other applications are wasting valuable system resources, restart Windows.

Explanation 4: In Windows 3.0, menus and menu items were allocated space from User's heap. In Windows 3.1, they are allocated space from a separate heap. This heap is for the exclusive use of menus and menu items.

Affected Functions: AppendMenu(), Insertmenu(), ModifyMenu(), CreateMenu(), CreatePopupMenu(), LoadMenu(), LoadMenuIndirect(), TrackPopupMenu(), GetSystemMenu() (when fRevert = FALSE).

Cause 5: There are no items in the menu.

Resolution 5: Use GetMenuItemCount() to make sure the menu is not empty.

Explanation 5: Nothing to be deleted or removed.

Affected Functions : RemoveMenu(), DeleteMenu().

Cause 6: The menu resource could not be found (FindResource()) or loaded

(LoadResource()) or locked (LockResource()) in memory.

Resolution 6: Ensure that the menu resource exists and that the hInst parameter refers to the correct hInstance. Try increasing the number of file handles using SetHandleCount() and increasing available global memory by closing some applications. For more information about the causes of failure of resource functions, query this Knowledge Base on the following keywords:

   failure and LoadResource and FindResource and LockResource.

Explanation 6: Finding, loading, and locking a resource involves use of file handles, global memory, and the hInstance that has the menu resource.

Affected Functions: LoadMenu(), LoadMenuIndirect()

Cause 7. TrackPopupMenu() is called while another popup menu is being tracked in the system.

Explanation 7: Only one popup menu can be tracked in the system at any given time.

Affected Function: TrackPopupMenu()

Cause 8. The hMenu that has been passed to TrackPopupMenu() has been deleted. The debug mode of Windows 3.1 sends the following message :

"Menu destroyed unexpectedly by WM_INITMENUPOPUP"

Explanation 8: Windows sends a WM_INITMENUPOPUP to the application and expects the menu to not be destroyed.

Affected Function: TrackPopupMenu()

Cause 9. MENUITEMTEMPLATEHEADER 's versionNumber field is non-zero.

Explanation 9: In Windows 3.0 and 3.1, this field should always be 0.

Affected Function: LoadMenuIndirect()

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMen

Reasons Why RegisterClass() and CreateWindow() Fail

PSS ID Number: Q65257

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The RegisterClass() and CreateWindow() functions fail when the system resources are used up. The percentage of free system resources reflects the amount of available space in the USER and GDI heaps within Windows. The smaller amount of free space is reported in the Program Manager's About box because if either heap fills up, functions fail.

Under Windows NT, the USER and GDI heap resources are practically unlimited. Under Windows 95, the USER and GDI heap resources are greater than Windows 3.1, but not as great as under Windows NT.

MORE INFORMATION

If the amount of free system resources remains low after the application is exited, it is more likely that the GDI heap is filling. The main reason for the GDI heap filling is that GDI objects that are created by the application are not deleted or destroyed when they are no longer needed, or when the program terminates. Windows does not delete GDI objects (pens, brushes, fonts, regions, and bitmaps) when the program exits. Objects must be properly deleted or destroyed.

NOTE: Win32-based applications cannot cause the USER or GDI heaps to overflow when they terminate, because the system will release the resources to maximize available resources.

The following are two situations that can cause the USER heap to get full:

1. Memory is allocated for "extra bytes" associated with window classes and windows themselves. Make sure that the cbClsExtra and cbWndExtra fields in the WNDCLASS structure are set to 0 (zero), unless they really are being used.

2. Menus are stored in the USER heap. If menus are added but are not destroyed when they are no longer needed, or when the application terminates, system resources will go down.

CreateWindow() will also fail under the following conditions:

1. Windows cannot find the window procedure listed in the
   CreateWindow() call. Avoid this by ensuring that each window
   procedure is listed in the EXPORTS section of the program's DEF
   file.

2. CreateWindow() cannot find the specified window class.

3. The hwndparent is incorrect (make sure to use debug Windows to see
   the RIPs).

4. CreateWindow() cannot allocate memory for internal structures in
   USER heap.

5. The application returns 0 (zero) to the WM_NCCREATE message.

6. The application returns -1 to the WM_CREATE message.

Additional reference words: 3.00 3.10 3.50 4.00
KBCategory: kbui
KBSubcategory: UsrWndw

Receive/Send Multicasts in Windows NT & Win95 Using WinSock

PSS ID Number: Q131978

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.5, and 3.51
- Microsoft Windows version 4.0

SUMMARY

This article describes how a host can become a member of a multicast group and receive and send multicast packets with Windows NT using the Windows Sockets (WinSock) interface.

This functionality is also available with Windows 95.

MORE INFORMATION

Sending IP Multicast Datagrams

IP multicasting is currently supported only on AF_INET sockets of type SOCK_DGRAM.

To send a multicast datagram, specify an IP multicast address with a range of 224.0.0.0 to 239.255.255.255 as the destination address in a sendto() call.

By default, IP multicast datagrams are sent with a time-to-live (TTL) of 1, which prevents them from being forwarded beyond a single subnetwork. The following code demonstrates how to change this functionality:

```
  int ttl = 7 ; // Arbitrary TTL value.
  setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, (char *)&ttl, sizeof(ttl))
```

Multicast datagrams with a TTL of 0 are not transmitted on any subnetwork. Multicast datagrams with a TTL of greater than one may be delivered to more than one subnetwork if there are one or more multicast routers attached to the first-hop subnetwork.

A multicast router does not forward multicast datagrams with destination addresses between 224.0.0.0 and 224.0.0.255, inclusive, regardless of their TTLs. This particular range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. A socket option is available to override the default for subsequent transmissions from a given socket. For example

```
  unsigned long addr = inet_addr("157.57.8.1");
  setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF,
            (char *)&addr, sizeof(addr))
```

where "addr" is the local IP address of the desired outgoing interface. An address of INADDR_ANY may be used to revert to the default interface. Note that this address might be different from the one the socket is bound to.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), by default, a copy of the datagram is looped back by the IP layer for local delivery. Under some versions of UNIX, there is an option available to disable this behavior (IP_MULTICAST_LOOP). This option is not supported in Windows NT. If you try to disable this behavior, the call fails with the error WSAENOPROTOOPT (Bad protocol option).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

Receiving IP Multicast Datagrams

Before a host can receive IP multicast datagrams, it must become a member of one or more IP multicast groups. A process can ask the host to join a multicast group by using the following socket option

```
   struct ip_mreq mreq;
```

where "mreq" is the following structure:

```
   struct ip_mreq {
       struct in_addr imr_multiaddr;   /* multicast group to join */
       struct in_addr imr_interface;   /* interface to join on    */
   }
```

For example:

```
        #define RECV_IP_ADDR   "225.6.7.8"  // arbitrary multicast address

        mreq.imr_multiaddr.s_addr = inet_addr(RECV_IP_ADDR);
        mreq.imr_interface.s_addr = INADDR_ANY;
        err = setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                        (char*)&mreq, sizeof(mreq))
```

Note that it is necessary to bind to an address before calling the setsockopt() function.

Every membership is associated with a single interface, and it is possible to join the same group on more than one interface. The address of "imr_interface" should be INADDR_ANY to choose the default multicast interface, or one of the host's local addresses to choose a particular (multicast-capable) interface.

The maximum number of memberships is limited only by memory and what the network card supports.

The following code sample can be used to drop a membership

```
   struct ip_mreq mreq;
   setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP,
              (char*)&mreq,sizeof(mreq))
```

where "mreq" contains the same values as used to add the membership. The memberships associated with a socket are also dropped when the socket is closed or the process holding the socket is killed. However, more than one socket may claim a membership in a particular group, and the host remains a member of that group until the last claim is dropped.

The memberships associated with a socket do not necessarily determine which datagrams are received by that socket. Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram; however, delivery of a multicast datagram to a particular socket is based on the destination port (or protocol type, for raw sockets), just as with unicast datagrams. To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified (that is, INADDR_ANY).

More than one process may bind to the same SOCK_DGRAM UDP port if the bind() call is preceded by the following code:

```
   int one = 1;
   setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *)&one, sizeof(one))
```

In this case, every incoming multicast or broadcast UDP datagram destined for the shared port is delivered to all sockets bound to the port.

The definitions required for the new, multicast-related socket options are located in the WINSOCK.H file. All IP addresses are passed in network byte-order.

REFERENCES
==========

The file DOC\MISC\MULTICAST.TXT is included with the Win32 SDK.

Additional reference words: 3.50 4.00
KBCategory: kbnetwork kbcode
KBSubcategory: NtwkWinsock

Redirection Issues on Windows 95 MS-DOS Apps and Batch Files

PSS ID Number: Q150956

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) version 3.5

SUMMARY
=======

Due to implementation differences on the Microsoft Windows 95 and Microsoft Windows NT platforms, special steps need to be taken to allow the same code to redirect the output of MS-DOS applications and batch (.BAT) files on both platforms.

MORE INFORMATION
================

When you redirect the output of an MS-DOS application or a batch file to a Win32 GUI or Console application, the intuitive method is to launch the MS-DOS process as a DETACHED_PROCESS with a redirected standard output handle to an anonymous pipe. The parent then uses the read end of the pipe to read the redirected output of the MS-DOS process.

This design works as expected on Windows NT. On Windows 95, however, this design causes the parent process to hang because the redirection pipe does not close when a redirected MS-DOS application or batch file exits.

A simple solution is to launch a hidden Win32 console application as an interface between the parent Win32 app and the MS-DOS child. The Win32 application spawns a "hidden" console application that inherits standard handles that have been redirected to an anonymous pipe. The hidden console application then spawns the MS-DOS application causing the MS-DOS application to inherit the hidden console. This behavior results in the MS-DOS application indirectly inheriting the redirected standard handles of the Win32 application.

SAMPLE CODE

The code below outlines a technique that works correctly on both Windows NT and Windows 95:

```
/*-----------------------Win32 application code----------------------------

This code redirects standard handles of the Win32 application and then
spawns a console application (CONSPAWN.EXE.) with a hidden window. CONSPAWN
inherits the redirected standard handles and spawns the application passed
to it on its Command line (DOSAPP.EXE.) When DOSAPP.EXE writes to its
STDOUT or STDERR handles, the output is redirected to the pipe created in
this Win32 application.
------------------------------------------------------------------------*/
```

```
      SECURITY_ATTRIBUTES sa          = {0};
      STARTUPINFO         si          = {0};
      PROCESS_INFORMATION pi          = {0};
      HANDLE              hPipeWrite  = NULL;
      HANDLE              hPipeRead   = NULL;
      CHAR Buffer[256];

      sa.nLength = sizeof(sa);
      sa.bInheritHandle = TRUE;
      sa.lpSecurityDescriptor = NULL;


      // Create pipe for output redirection.
      CreatePipe(&hPipeRead,  // read handle
              &hPipeWrite, // write handle
              &sa,      // security attributes
              0      // number of bytes reserved for pipe - 0 default
              );

      // Create pipe for input redirection. In this code, you do not
      // redirect the output of the child process, but you need a handle
      // to set the hStdInput field in the STARTUP_INFO struct. For safety,
      // you should not set the handles to an invalid handle.

      CreatePipe(&Read2,   // read handle
              &hWrite2,    // write handle
              &sa,      // security attributes
              0      // number of bytes reserved for pipe - 0 default
              );

      // Make child process use hPipeWrite as standard out, and make
      // sure it does not show on screen.
      si.cb = sizeof(si);
      si.dwFlags     = STARTF_USESHOWWINDOW | STARTF_USESTDHANDLES;
      si.wShowWindow = SW_HIDE;
      si.hStdInput   = hWrite2;
      si.hStdOutput  = hPipeWrite;
      si.hStdError   = hPipeWrite;

      CreateProcess (
NULL,
            "CONSPAWN.EXE DOSAPP.EXE",
            NULL, NULL,
            TRUE, 0,
            NULL, NULL,
            &si, &pi);

       /* now that this has been inherited, close it to be safe.
          You don't want to write to it accidentally*/
      CloseHandle(hPipeWrite);

      while(TRUE)
      {
         bTest=ReadFile(
hRead,       // handle of the read end of our pipe
```

```
            &Buffer,     // address of buffer that receives data
            255,         // number of bytes to read
            &NumberOfBytesRead,// address of number of bytes read
            NULL         // non-overlapped.
            );

         if (!bTest){
            printf("Error #%d reading pipe.",GetLastError());
            return 0;
         }

         if (NumberOfBytesRead == 0){
            // pipe done - normal exit path.
            break;
         }
         // do something with data.
      }

      // Wait for CONSPAWN to finish.
      WaitForSingleObject (pi.hProcess, INFINITE);

      // Close all remaining handles
      CloseHandle (pi.hProcess);
      CloseHandle (hPipeRead);
      }

/*----------Console application (CONSPAWN.EXE) code-----------------------

This program (CONSPAWN) is launched with a hidden console that inherits the
redirected standard handles of the Win32 application. The application that
CONSPAWN launches in the same hidden console inherits the same redirected
standard handles. This behavior redirects the standard handles of the
MS-DOS application to be launched to the pipe created in the parent Win32
application.
------------------------------------------------------------------------*/

    #include <windows.h>
    #include <stdio.h>

    void main (int argc, char *argv[])
    {
        BOOL                bRet = FALSE;
        STARTUPINFO         si   = {0};
        PROCESS_INFORMATION pi   = {0};

        // Make child process use this app's standard files.
        si.cb = sizeof(si);
        si.dwFlags    = STARTF_USESTDHANDLES;
        si.hStdInput  = GetStdHandle (STD_INPUT_HANDLE);
        si.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);
        si.hStdError  = GetStdHandle (STD_ERROR_HANDLE);

        bRet = CreateProcess (NULL, argv[1],
                              NULL, NULL,
                              FALSE, 0,
```

```
                            NULL, NULL,
                            &si, &pi
                            );

        if (bRet)
        {
            WaitForSingleObject (pi.hProcess, INFINITE);
            CloseHandle (pi.hProcess);
            CloseHandle (pi.hThread);
        }
    }
```

Additional reference words: 4.00
KBCategory: kbprg kbhowto
KBSubcategory: BseCon

Reducing the Count on a Semaphore Object

PSS ID Number: Q94997

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

ReleaseSemaphore(), which increments the count of a given semaphore object by a specified amount, will not take a negative value.

If, for some reason, you want to reduce the number of available semaphore "slots" to temporarily restrict or reduce access, you may loop calling WaitForSingleObject() with a zero timeout, counting the number of times it succeeds. When you no longer need to hold the semaphore slots, call ReleaseSemaphore() with the number of slots counted.

Note that this method does not prevent other threads from taking a semaphore slot when your thread is looping.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

Registering Multiple RPC Server Interfaces

PSS ID Number: Q129975

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.5

When registering multiple server interfaces from multiple threads or from a single thread under the same process, the RpcServerListen() function should be called only once.

An application process may register two completely separate Remote Procedure Call (RPC) server interfaces from two separate threads or from a single thread. However, when doing so, the RpcServerListen() function should be called only once from any thread.

RPC APIs are called on a per process basis. From the perspective of RPC run times, a process can register multiple interfaces using one or more protocol sequences, but must call the RpcServerListen() function only once from any one thread. Calling the RpcServerListen() function more than once results in the run time generating an exception called RPC_S_ALREADY_LISTENING.

Additional reference words: 3.50
KBCategory: kbnetwork kbnetwork
KBSubcategory: NtwkRpc

RegSaveKey() Requires SeBackupPrivilege

PSS ID Number: Q106383

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

The description for RegSaveKey() states the following:

> The caller of this function must possess the SeBackupPrivilege security privilege.

This means that the application must explicitly open a security token and enable the SeBackupPrivilege. By granting a particular user the right to back up files, you give that user the right only to gain access to the security token (that is, the token is not automatically created for the user but the right to create such a token is given). You must add additional code to open the token and enable the privilege.

MORE INFORMATION
================

The following code demonstrates how to enable SeBackupPrivilege:

```
   static HANDLE           hToken;
   static TOKEN_PRIVILEGES tp;
   static LUID             luid;

   // Enable backup privilege.

   OpenProcessToken( GetCurrentProcess(),
      TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken ) ;
   LookupPrivilegeValue( NULL, "SeBackupPrivilege", &luid );
   tp.PrivilegeCount           = 1;
   tp.Privileges[0].Luid       = luid;
   tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
   AdjustTokenPrivileges( hToken, FALSE, &tp,
      sizeof(TOKEN_PRIVILEGES), NULL, NULL );

   // Insert your code here to save the registry keys/subkeys.

   // Disable backup privilege.

   AdjustTokenPrivileges( hToken, TRUE, &tp, sizeof(TOKEN_PRIVILEGES),
      NULL, NULL );
```

Note that you cannot create a process token; you must open the existing process token and adjust its privileges.

The DDEML Clock sample has similar code sample at the end of the CLOCK.C file where it obtains the SeSystemTimePrivilege so that it can set the system time.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Removing Focus from a Control When Mouse Released Outside

PSS ID Number: Q66947

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Under normal circumstances, when you move the mouse cursor into the client area of a child-window control, click it, and then release the mouse button, the child window sends a WM_COMMAND message to its parent and retains the focus.

If you move the mouse into the client area of the child-window control, press the mouse button, move the mouse cursor out of the client area of the control, and then release the mouse button, the control does not send a WM_COMMAND message. However, the control retains the focus.

If you do not want the control to retain the focus, you can remove it by performing the following steps:

1. Define a static Boolean flag in the parent window function.

2. When a WM_PARENTNOTIFY message is received, set the flag to TRUE. This indicates that the mouse button has been pressed while the mouse cursor is in the client area of the control.

3. If a WM_COMMAND message is received, reset the flag to FALSE and perform normal processing.

4. Otherwise, if a WM_MOUSEMOVE message is received, the mouse button was released after the mouse cursor was moved outside the control. Reset the flag to FALSE, and use SetFocus() to move the focus to the desired window.

MORE INFORMATION

When the mouse cursor is in the client area of a control and you press the mouse button, the parent window will receive a WM_PARENTNOTIFY message and a WM_MOUSEACTIVATE message. A Boolean (BOOL) flag should be set when the message is processed to indicate that this occurred.

The parent window will receive other messages, including a number of WM_CTLCOLOR messages, when the mouse is moved around with the mouse button down. When the mouse button is released, the parent window receives only

one of two messages:

1. WM_COMMAND: The mouse button was released over the control.

2. WM_MOUSEMOVE: The mouse button was released outside the control.

Note that these are not the only messages received when the button is released, but these two are mutually exclusive.

In response to either message, the following steps must take place:

1. Reset the flag indicating a mouse press.

2. Call SetFocus() or send a WM_KILLFOCUS to the control in question to move the focus as desired.

If WM_KILLFOCUS is used, the ID of the control or its handle must be known. SetFocus(NULL) or SetFocus(hWndParent) removes the focus from the control but does not set the focus to any other control in the window.

In a dialog box, SetFocus(NULL) MUST be used. SetFocus(hDlg) does not remove the focus from the button.

The following code sample is taken from the dialog box procedure of a dialog that has a single OK button. If the mouse button is pressed while the mouse cursor is over the button, the mouse is moved outside the button, and then the mouse button is released, the focus is removed from the OK button.

```
BOOL FAR PASCAL AboutProc(HWND hDlg, unsigned iMessage,
                          WORD wParam, LONG lParam)
    {
    static BOOL fMousePress;

    switch (iMessage)
        {
        case WM_INITDIALOG:
            fMousePress = FALSE;
            return TRUE;

        case WM_PARENTNOTIFY:  // or WM_MOUSEACTIVATE
            fMousePress = TRUE;
            break;

        case WM_MOUSEMOVE:
            if (fMousePress)
                SetFocus(NULL);
            fMousePress = FALSE;
            break;

        // Only command is the OK button.
        case WM_COMMAND:
            if (wParam == IDOK)
                EndDialog(hDlg, TRUE);
            break;
```

```
        }
    return FALSE;
    }
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbcode
KBSubcategory: UsrCtl

Replace IsTask() with GetExitCodeProcess()

PSS ID Number: Q108228

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

In Windows 3.1, the application programming interface (API) IsTask() can be used to determine whether a process is still running or whether it has terminated. As the help file indicates, this function is obsolete in the Win32 API.

To get this functionality through the Win32 API, use the API GetExitCodeProcess(). This function takes the handle as the first parameter and returns the exit code or STILL_ACTIVE in the second parameter:

```
BOOL GetExitCodeProcess(hProcess, lpdwExitCode)

HANDLE hProcess;
LPDWORD lpdwExitCode;
```

As an alternative, you can also use WaitForSingleObject(). Pass the process handle as the first parameter and a timeout value for the second parameter:

```
DWORD WaitForSingleObject(hObject, dwTimeout)

HANDLE hObject;
DWORD dwTimeout;
```

The process handle is signaled when the process terminates. Pass in 0 (zero) for the timeout if you would like to poll or start another thread to wait with an INFINITE timeout value.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseProcThrd

Replacing the Shell (Program Manager)

PSS ID Number: Q100328

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

=======

To replace the current shell, change the following registry key:

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
   Microsoft\
   Windows NT\
   CurrentVersion\
   Winlogon\
   Shell
```

Note that Program Manager combines the functionality of Program Manager and Task Manager (the Task Manager installed is not actually run). Therefore, you must take this into account. In Windows NT 3.1, if the new shell does not replace the Task Manager functionality, the replacement string should contain both the new shell name and TASKMAN.EXE, separated by commas. In Windows NT 3.5, the new shell should either spawn TASKMAN.EXE or your own task manager, specified in

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
   Microsoft\
   Windows NT\
   CurrentVersion\
   Winlogon\
   Taskman
```

The value does not exist by default, it must be added. The value type is REG_SZ.

To update the string that is retrieved when you call GetPrivateProfileString(), change the string in the following registry key:

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
   MICROSOFT\
   Windows NT\
   Current Version\
   WOW\
   Boot\
   Shell
```

The duplicate entry is for compatibility with Windows 3.1.

MORE INFORMATION
================

WritePrivateProfileString() changes the following registry key:

```
   HKEY_LOCAL_MACHINE\
      SOFTWARE\
      Microsoft\
      Windows NT\
      CurrentVersion\
      WOW\
      Boot\
      Shell
```

It does not have the desired effect of actually changing the Shell.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Replacing the Windows NT Task Manager

PSS ID Number: Q89373

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

Under Windows 3.1, the Task Manager is an easily replaceable program. However, under the Win32 subsystem of Windows NT, it is very difficult to replace the Task Manager, due to special programming considerations. In general, it is not recommended that users attempt this.

MORE INFORMATION

================

Special requirements for the Task Manager make it very different from the Windows 3.1 Task Manager. The EndTask button handling is done through internal application programming interface (API) functions. These API functions are not documented. The situation is the same for handling foreground management, hung applications, and priority issues (to make sure that the Task Manager will come up as fast as possible). In addition, the Windows NT Task Manager uses shortcut ("hot") keys.

In Windows NT 3.1, the Task List has been incorporated into the Program Manager. To remove the Task List, you must also remove the Program Manager. The full functionality of the Task List (as found in TASKMAN.EXE) is now folded into the Program Manager (PROGMAN.EXE). If you completely delete the TASKMAN.EXE file from your system and from the registry location

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
   Microsoft\
   Windows NT\
   CurrentVersion\
   WinLogon\
   Shell
```

you will still be able to invoke the Task List because it is built into the Program Manager.

In Windows NT 3.5, Program Manager checks the registry for a Taskman entry. If the Taskman entry is found, Program Manager will launch the application, instead of using its built-in Taskman. The registry entry is:

```
HKEY_LOCAL_MACHINE\
   SOFTWARE\
   Microsoft\
   Windows NT\
```

```
      CurrentVersion\
      Winlogon\
      Taskman
```

This entry does not not exist by default. You will have to create this value, with type REG_SZ.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Replacing Windows NT Control Panel's Mouse Applet

PSS ID Number: Q110704

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

The Control Panel includes a Mouse applet as a standard applet that is shipped with the system. In Windows 3.1, this applet can be overridden by an ISV/OEM's mouse driver or module. Windows 3.1 accomplished this by the Control Panel doing a GetModuleHandle() call on "Mouse". If MOUSE.DLL or MOUSE.EXE was already loaded in the system, Control Panel would look for the entry point "CPlApplet". If found, Control Panel would send the following messages:

   CPL_NEWINQUIRE

   -or-

   CPL_INQUIRE (the former is preferred)

This would return the icon and strings to replace the mouse icon already in Control Panel.

In Windows NT, because of the separation of Kernel drivers from applications by the client-server interface, the GetModuleHandle() call does not work. Consequently, the same functionality must be achieved in a slightly different way. The Control Panel calls LoadLibrary("Mouse") to look for a MOUSE.DLL or a MOUSE.EXE. If this call fails, no other checks are made.

If LoadLibrary() succeeds, the Control Panel looks for the "CPlApplet" entry point, sends a CPL_INIT message, and then sends a CPL_NEWINQUIRE. If CPL_NEWINQUIRE fails, a CPL_INQUIRE is sent; however, it is preferable to have the applet implement the newer CPL_NEWINQUIRE message. The string information returned by the CPL_NEWINQUIRE message can be in either UNICODE or ANSI (UNICODE is preferred) as long as the dwSize field is set correctly. See the CPL.H public header file for these messages and structures (for example CPLINFOW or CPLINFOA, where CPLINFOW is the default).

When the User double-clicks the Mouse applet icon, the MOUSE.DLL or MOUSE.EXE will receive a CPL_DBLCLK message from the CPlApplet interface. This routine must return TRUE to the Control Panel if the routine runs its own dialog. If FALSE is returned, the internal Mouse Dialog box will be presented to the User.

The Control Panel will send the CPL_EXIT message to the Mouse applet when it wants to unload the module or terminate. The applet must use this message to perform tasks such as calling UnRegisterWindowClass(), freeing memory, and unloading DLLs.

NOTE: It is not possible to replace any of the other standard Windows NT 3.1 Control Panel applets. As of Windows NT 3.5, it is also possible to replace the Keyboard applet in the same manner.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrMisc

Resource Sections are Read-only

PSS ID Number: Q126630

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.2

SUMMARY
=======

Resource sections are read-only by default under Windows NT and Win32s. However, under Windows NT, the resource section may appear to be read/write. If an application tries to write to the resource section, an exception occurs. Windows NT handles the exception by duplicating the page and making it read/write. Therefore, under Windows NT it is possible to write to the resource section, even though its section attribute is read-only.

In Win32s version 1.2, the resource section is read-only and you cannot write to it. To work around this, link with /SECTION:.rsrc,rw to make the resource section read/write. Or copy the resource to your own buffer and work with it from there. You cannot modify the protection of the resource section because the memory is owned by the system.

In Win32s version 1.25a and later, the resource section is read/write, regardless of what is specified in the section attributes.

Windows 95 has a handler similar to the one used in Windows NT.

MORE INFORMATION
================

Under Windows NT, the default top level handler detects writes to resources and will make the resource writable. If you are running outside of a debugger and you have no exception handler, your resource writes will silently work. If you are running under the debugger, your resource write will look like an access violation:

```
   First-Chance Exception in msin32.exe: 0xC0000005: Access Violation
```

This allows you to "fix" your resource writes. If you have the debugger pass on the exception to your application and you have no handler, the default handler will make your resource writable.

The disadvantage of setting the attribute of the resource section to read/write is that Windows NT and Windows 95 will use a separate copy of the resource section for each process that uses this section, instead of one copy for all processes.

Additional reference words: 1.20 3.50 4.00 LoadResource LockResource
KBCategory: kbui
KBSubcategory: UsrRes W32s

Restriction on Named-Pipe Names

PSS ID Number: Q100291

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

Named pipes are implemented in Windows NT using the same approach used for file systems. Within Windows NT, named pipes are represented as file objects.

During the design phase, one idea for the implementation was to allow sub directories of named pipes. For example, a developer could create a named pipe sub directory called \MYPIPES. It would then be possible to create and use pipes called \MYPIPES\PIPE1 and \MYPIPES\PIPE2, but it would not be possible to use \MYPIPES as a pipe.

In the end, this idea was not implemented, so sub directories are not supported. This does have some effect on the named-pipe names that are allowed. If a pipe named \MYPIPES is created, it is not possible to subsequently create a pipe named \MYPIPES\PIPE1, because \MYPIPES is already a pipe name and cannot be used as a sub directory. It is possible to
create a pipe named \MYPIPES\PIPE1, but only if there is no pipe named \MYPIPES. The error received is ERROR_INVALID_NAME (123).

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseIpc

Results of GetFileInformationByHandle() Under Win32s

PSS ID Number: Q123813

The information in this article applies to:

- Microsoft Win32s versions 1.2 and 1.25a

The GetFileInformationByHandle() function returns information about the given file in a structure of type BY_HANDLE_FILE_INFORMATION.

The following are the BY_HANDLE_FILE_INFORMATION structure fields and the values they contain under Win32s:

   dwFileAttributes - Always 0 in version 1.2 (A bug that has been fixed.)
   ftCreationTime - Always 0. (An MS-DOS file system limitation.)
   ftLastAccessTime - Always 0. (An MS-DOS file system limitation.)
   ftLastWriteTime - Correct value.
   dwVolumeSerialNumber - Always 0. (A Win32s limitation.)
   nFileSizeHigh - Correct value.
   nFileSizeLow - Correct value.
   nNumberOfLinks - Always 1.
   nFileIndexHigh - Always 0. (A Win32s limitation.)
   nFileIndexLow - Always 0. (A Win32s limitation.)

Additional reference words: 1.20
KBCategory: kbprg
KBSubcategory: W32s

Retrieving Counter Data from the Registry

PSS ID Number: Q107728

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

=======

The performance data begins with a structure of type PERF_DATA_BLOCK and is followed by PERF_DATA_BLOCK.NumObjectTypes data blocks. Each data block begins with a structure of type PERF_OBJECT_TYPE, followed by PERF_OBJECT_TYPE.NumCounters structures of type PERF_COUNTER_DEFINITION. Next, there are PERF_OBJECT_TYPE.NumInstances structures of type PERF_INSTANCE DEFINITION, each directly followed by an instance name, a structure of type PERF_COUNTER_BLOCK and PERF_OBJECT_TYPE.NumCounters counters. All of these data types are described in WINPERF.H.

MORE INFORMATION

================

The following steps are used to retrieve all of the counter information from the registry:

1. Allocate a buffer to obtain the performance data. For single objects, you may need at little as 1.5K. For global objects, you may need as much as 50K. Call RegQueryValueEx() to obtain the data. If the call returns ERROR_MORE_DATA, then the buffer size was not big enough. Increase the size of the buffer and try again. Repeat until the call is successful.

2. Get the first object type in PerfObj.

3. Get the first instance.

4. Get the first counter and its data. At this point, if the object has no instances, the next thing will be a pointer to a single PERF_COUNTER_BLOCK; otherwise, the next thing will be a pointer to the first PERF_INSTANCE_DEFINITION. Check PerfObj->NumInstances to find out how many instances there are.

5. Get the next counter and its data. Repeat for all PerfObj->NumCounters counters.

6. After all counters are retrieved for the instance, get the next instance and all its counters. Repeat for all PerfObj->NumInstances instances.

7. After all instances of the object type are retrieved, move to the next object type and repeat steps 3 - 7.

Sample Code

```
#include <windows.h>
#include <stdio.h>
#include <malloc.h>

#define TOTALBYTES    8192
#define BYTEINCREMENT 1024

void main()
{
   PPERF_DATA_BLOCK PerfData = NULL;
   PPERF_OBJECT_TYPE PerfObj;
   PPERF_INSTANCE_DEFINITION PerfInst;
   PPERF_COUNTER_DEFINITION PerfCntr, CurCntr;
   PPERF_COUNTER_BLOCK PtrToCntr;
   DWORD BufferSize = TOTALBYTES;
   DWORD i, j, k;

// Allocate the buffer.
   PerfData = (PPERF_DATA_BLOCK) malloc( BufferSize );

   while( RegQueryValueEx( HKEY_PERFORMANCE_DATA,
                           "Global",
                           NULL,
                           NULL,
                           (LPBYTE) PerfData,
                           &BufferSize ) == ERROR_MORE_DATA )
   {
   // Get a buffer that is big enough.
      BufferSize += BYTEINCREMENT;
      PerfData = (PPERF_DATA_BLOCK) realloc( PerfData, BufferSize );
   }

// Get the first object type.
   PerfObj = (PPERF_OBJECT_TYPE)((PBYTE)PerfData +
      PerfData->HeaderLength);

// Process all objects.
   for( i=0; i < PerfData->NumObjectTypes; i++ )
   {
      printf( "\nObject: %ld\n", PerfObj->ObjectNameTitleIndex );

   // Get the counter block.
      PerfCntr = (PPERF_COUNTER_DEFINITION) ((PBYTE)PerfObj +
                 PerfObj->HeaderLength);

      if( PerfObj->NumInstances > 0 )
      {
      // Get the first instance.
         PerfInst = (PPERF_INSTANCE_DEFINITION)((PBYTE)PerfObj +
                    PerfObj->DefinitionLength);
```

```
      // Retrieve all instances.
         for( k=0; k < PerfObj->NumInstances; k++ )
         {
            printf( "\n\tInstance: %S\n", (char *)((PBYTE)PerfInst +
                    PerfInst->NameOffset) );
            CurCntr = PerfCntr;

         // Get the first counter.
            PtrToCntr = (PPERF_COUNTER_BLOCK)((PBYTE)PerfInst +
                        PerfInst->ByteLength);

         // Retrieve all counters.
            for( j=0; j < PerfObj->NumCounters; j++ )
            {
               printf("\t\tCounter: %ld\n",CurCntr->CounterNameTitleIndex);
            // Data is (LPVOID)((PBYTE)PtrToCntr + CurCntr->CounterOffset);

            // Get next counter.
               CurCntr =  (PPERF_COUNTER_DEFINITION)((PBYTE)CurCntr +
                          CurCntr->ByteLength);

            }

         // Get the next instance.
            PerfInst = (PPERF_INSTANCE_DEFINITION)((PBYTE)PtrToCntr +
                       PtrToCntr->ByteLength);
         }
      }
      else
      {
      // Get the first counter.
         PtrToCntr = (PPERF_COUNTER_BLOCK) ((PBYTE)PerfObj +
                     PerfObj->DefinitionLength );

      // Retrieve all counters.
         for( j=0; j < PerfObj->NumCounters; j++ )
         {
            printf( "\tCounter: %ld\n", PerfCntr->CounterNameTitleIndex );

         // Data is (LPVOID)((PBYTE)PtrToCntr + PerfCntr->CounterOffset);

            PerfCntr = (PPERF_COUNTER_DEFINITION)((PBYTE)PerfCntr +
                       PerfCntr->ByteLength);
         }
      }

      // Get the next object type.
      PerfObj = (PPERF_OBJECT_TYPE)((PBYTE)PerfObj +
                PerfObj->TotalByteLength);
   }
}
```

```
 +
                PerfObj->TotalByteLength);
   }
}
```

Note that the instance names are retrieved in a fashion that is similar to retrieving the data.

The steps above showed how to obtain all of the counters. You can retrieve only the counters that pertain to a particular object by using the titles database. The information is stored in the registry in the format index, name, index, name, and so forth.

To retrieve the titles database and store it in TitlesDatabase:

1. Open the key:

```
      RegOpenKeyEx( HKEY_LOCAL_MACHINE,
         "Software\\Microsoft\\Windows NT\\CurrentVersion\\Perflib\\009",
                0,
         KEY_READ,
            &Hkey);
```

   Note that 009 is a language ID, so this value will be different on a non-English version of the operating system.

2. Query the information from the key:

```
      RegQueryInfoKey(
                  Hkey,
         (LPTSTR) Class,
             &ClassSize,
                  NULL,
               &Subkey,
              MaxSubKey,
              &MaxClass,
                &Values,
               &MaxName,
               &MaxData,
               &SecDesc,
          &LastWriteTime );
```

3. Allocate a buffer to store the information:

```
      TitlesDataBase = (PSTR) malloc( (MaxData+1) * sizeof(TCHAR) )
```

4. Retrieve the data:

```
      RegQueryValueEx(     Hkey,
            (LPTSTR) "Counters",
                           NULL,
                           NULL,
        (LPBYTE) TitlesDataBase,
                       &MaxData );
```

Once you have the database, it is possible to write code that will go through all objects, searching by index (field ObjectNameTitleIndex) or by type (field ObjectNameTitle - which is initially NULL).

Or, you could obtain only the performance data for specified objects by changing the call to ReqQueryValueEx() in step 1 of the SUMMARY section above to:

```
   RegQueryValueEx( HKEY_PERFORMANCE_DATA,
                           Indices,
                              NULL,
                              NULL,
                         PerfData,
                       &BufferSize );
```

Note that the only difference here is that instead of specifying "Global" as the second parameter, you specify a string that represents the decimal value(s) for the object(s) of interest that are obtained from the titles database.

REFERENCES
==========

The PVIEWER and PERFMON samples in the MSTOOLS\SAMPLES\SDKTOOLS directory contain complete sample code that deals with performance data.

For more information, please see the "Performance Overview" in the Win32 SDK documentation and the volume titled "Optimizing Windows NT" in the Windows NT Resource Kit.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Retrieving DIBs from the Clipboard

PSS ID Number: Q106386

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Retrieving a DIB (device-independent bitmap) from the clipboard can take significantly more time than retrieving a bitmap from the clipboard. The difference stems from the fact that a bitmap is a GDI object and a DIB is a global memory object.

MORE INFORMATION
================

When SetClipBoardData() is passed a global memory handle, as it is when it is passed a handle to a DIB, all the data gets copied into the Win32 server and put into a sharable section of memory. When the DIB is retrieved with GetClipBoardData(), the shared memory is mapped into the application's virtual address space and the memory handle is cached. Any subsequent calls to GetClipBoardData() return quickly, because the memory does not have to be remapped.

In contrast, when retrieving a bitmap with GetClipBoardData(), only a handle is created, because a bitmap is a GDI object.

When CloseClipboard() is called, all of the cached handles to shared memory and GDI objects are deleted.

Rather than reopening the clipboard, it is a good idea to keep a local copy of anything retrieved from the clipboard if the item will be used again after the clipboard has been closed. In general, data should be retrieved from the clipboard only when the application is doing a paste or if the application is a clipboard viewer processing a WM_DRAWCLIPBOARD message.

The data for a GDI object exists on the server side. In other words, bitmaps and DDBs (device-dependent bitmaps) exist in the Win32 subsystem address space. Only the handles of GDI objects are private to an application. Therefore, to make a bitmap or a DDB accessible to another application, only a call to DuplicateHandle() is needed.

Note that even though it is faster to retrieve a DDB from the clipboard, it is still recommended to put a DIB on the clipboard rather than a DDB.

Additional reference words: 3.10 3.50 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrClp

Retrieving Font Styles Using EnumFontFamilies()

PSS ID Number: Q84131

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Windows version 3.1 introduces the concept of a font style. In previous versions of Windows, a font could have the bold, italic, underline, and strikeout properties, which were supported by respective members in the LOGFONT and TEXTMETRIC structures. Windows 3.1 also supports these properties, as well as a style name for TrueType fonts. The article describes how to obtain the font style name during font enumeration, using the EnumFontFamilies function. For more information about obtaining style information without enumerating the fonts, query on the following words in the Microsoft Knowledge Base:

prod(winsdk) and getoutlinetextmetrics

MORE INFORMATION

In Windows 3.1, "style" refers to the weight and slant of a font. Windows supports a wide range of weights in the lfWeight member of the LOGFONT structure. (Two examples of weights are FW_BOLD, which is defined as 700, and FW_THIN, which is defined as 100). Very few applications, however, use any weights other than FW_BOLD and FW_DONTCARE (defined as 0).

Windows 3.1 builds on the support presently in Windows for these variations in weight and slant. Style names are NOT used in the LOGFONT structure except when the fonts are enumerated with EnumFontFamilies.

The ChooseFont dialog box in the common dialog boxes dynamic-link library (COMMDLG.DLL) demonstrates how style names are used. The ChooseFont dialog box has two list boxes: Font and Font Style. The Font list box lists the face names for all fonts installed and the Font Style list box lists the font styles for the currently selected face. For example, if any non-TrueType font (such as MS Sans Serif) is selected, the following styles appear in the Font Style list box:

Regular

Bold
Italic
Bold Italic

TrueType fonts may have these or more elaborate styles. For example, the "Lucida Sans" face includes the following style names:

Regular
Italic
Demibold Roman
Demibold Italic

In the case of Lucida Sans with the style of Demibold Roman or Demibold Italic, the lfWeight value is 600 (FW_DEMIBOLD).

In Windows 3.1, the EnumFontFamilies function can be used to obtain the style name of a font during font enumeration. The EnumFontFamilies function works in a manner very similar to the Windows 3.0 EnumFonts function.

EnumFontFamilies is prototyped as:

```
int EnumFontFamilies(HDC hdc, LPCSTR lpszFamily,
                     FONTENUMPROC lpfnEnumProc, LPARAM lpData)
```

The lpszFamily parameter points to a null-terminated string that specifies the family name (or typeface name) of the desired fonts. If this parameter is NULL, EnumFontFamilies selects and enumerates one font of each available font family. For example, to enumerate all fonts in the "Arial" family, lpszFamily points to a string buffer containing "Arial."

The following table illustrates the meanings of the terms, "typeface name," "font name," and "font style:"

Typeface Name	Font Name	Font Style
Arial	Arial	Regular
	Arial Bold	Bold
	Arial Italic	Italic
	Arial Bold Italic	Bold Italic
Courier New	Courier New	Regular
	Courier New Bold	Bold
	Courier New Italic	Italic
	Courier New Bold Italic	Bold Italic
Lucida Sans	Lucida Sans	Regular
	Lucida Sans Italic	Italic
	Lucida Sans Demibold Roman	Demibold Roman
	Lucida Sans Demibold Italic	Demibold Italic
MS Sans Serif	MS Sans Serif	Regular
	MS Sans Serif	Bold

MS Sans Serif	Italic
MS Sans Serif	Bold Italic

The first three typefaces in the above table are TrueType faces, the remaining typeface is MS Sans Serif. The typeface name is also sometimes referred to as the family name.

When dealing with non-TrueType fonts, typeface name and font name are the same. However, it is important to recognize the distinction when dealing with a TrueType font.

For example, CreateFont takes a pointer to a string containing the typeface name of the font to create. It is not valid to use Arial Bold as this string because Arial is a TrueType font and Arial Bold is a font name, not a typeface name.

If EnumFontFamilies is called with the lpszFamily parameter pointing to a valid TrueType typeface name, the callback function, which is specified in fntenmprc, will be called once for each font name for that typeface name. For example, if EnumFontFamilies is called with lpszFamily pointing to Lucida Sans, the callback function will be called four times; once for each font name.

If the lpszFamily parameter points to the typeface name of a non-TrueType font, such as MS Sans Serif, the callback will be called once for each face size supported by the font. The number of face sizes supported by the font can vary from font to font and from device to device. Note that the callback is called for different sizes, not for different styles. This behavior is identical to that found using the EnumFonts function.

Remember that, because TrueType fonts are continuously scalable, there is no reason for the callback function to be called for each size. If the callback function was called for each size that a TrueType font supported, the callback function would be called an infinite number of times!

The EnumFontFamilies callback function is prototyped as follows:

```
int CALLBACK EnumFontFamProc(LPNEWLOGFONT lpnlf,
                             LPNEWTEXTMETRIC lpntm,
                             int FontType, LPARAM lpData)
```

The lpnlf parameter points to a LOGFONT structure that contains information about the logical attributes of the font. If the typeface being enumerated is a TrueType font [(nFontType | TRUETYPE_FONTTYPE) is TRUE], this LOGFONT structure will have two additional members appended to the end of the structure, as follows:

```
   char   lfFullName[LF_FACESIZE*2];
   char   lfStyleName[LF_FACESIZE];
```

It is important to remember that these two additional fields are used *only* during enumeration with EnumFontFamilies and nowhere else in Windows. The documentation for the EnumFontFamilies function on pages

266-268 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 2: Functions" manual refers to the NEWLOGFONT structure which contains the additional members listed above. However, the NEWLOGFONT structure is not defined in the WINDOWS.H header file. To address this situation, use the ENUMLOGFONT structure which is defined in the WINDOWS.H file but is not listed in the Windows SDK documentation.

To retrieve the style name and full name of a font without using enumeration, use the GetOutlineTextMetrics function.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiTt

Retrieving Handles to Menus and Submenus

PSS ID Number: Q67688

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

To change the contents of a menu, you must have its handle. The handle of the menu associated with a given window is available through the GetMenu() function.

To obtain a reference to a particular text item in the menu, use the GetMenuString() function. The definition for this function is

```
   GetMenuString(hMenu, wIDItem, lpString, nMaxCount, wFlag)
```

where

```
   hMenu     = The menu handle
   wIDItem   = The ID of the item or the zero-based offset of the
               item within the menu
   lpString  = The buffer that is to receive the text
   nMaxCount = The length of the buffer
   wFlag     = MF_BYCOMMAND or MF_BYPOSITION
```

If a menu item has a mnemonic, the text will contain an ampersand (&) character preceding the mnemonic character.

To obtain the handle to a submenu of the menu bar, use the GetSubMenu() function. The second parameter, nPos, is the zero-based offset from the beginning of the menu.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMen

Retrieving Palette Information from a Bitmap Resource

PSS ID Number: Q124947

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s versions 1.2, 1.25a, and 1.3

SUMMARY

You may sometime need to create a logical palette from a bitmap resource in order to display the bitmap with the maximum number of available colors. For example, on an 8 bit-per-pixel display, a logical palette is necessary to draw a 256-color bitmap on a device context for that display. The LoadBitmap function does not return or take a palette as one of its parameters; thus, for example, there is no way to incorporate a palette with a 256-color bitmap loaded with LoadBitmap. Therefore, an application must load the resource as a device-independent bitmap (DIB), rather than a device-dependent bitmap (DDB), in order to retrieve the bitmap's color information. An application can use the FindResource, LoadResource, and LockResource functions to do this.

The solution differs slightly for Win32s.

MORE INFORMATION

Windows and Windows NT

A bitmap (.BMP file) is stored in an application's resources as a (DIB), along with a color table if one exists. When a DIB is loaded from an application's resources with the LoadBitmap function, a DDB is returned. This DDB is a bitmap compatible with the screen. Routines such as CreateDIBitmap and SetDIBits that convert DIBs to DDBs take a handle to a device context as their first parameter. This tells the routine what kind of DDB to create. If this device context currently has a palette selected into it, then CreateDIBitmap or SetDIBits can use this palette to create the DDB. Without a palette, the routines are restricted to system colors when matching the DIB's colors to the DDB's colors. For example, on an 8 bit-per-pixel display, the resulting DDB can have only up to 20 different colors. With a logical palette, the resulting bitmap could have had up to 256 different colors.

If the bitmap is loaded as a DIB from the resource, then an application can

query the DIB's color table and create a logical palette for the DIB. Then, it can call either CreateDIBitmap or SetDIBits, along with a device context with that palette selected, to obtain a DDB compatible with that palette. To load a bitmap from a resource as a DIB, you can use the FindResource function with the RT_BITMAP flag set and then use the LoadResource function to load it. You can lock the resource with the LockResource function.

The following code demonstrates how to use the above technique to load a DIB from an application's resources, create a palette for it, and then create a DDB out of it. The LoadResourceBitmap function below can be used in place of the LoadBitmap function. The only additional parameter needed is the address of a logical palette handle. The palette handle referenced will contain a handle to a logical palette after the function is called.

```
HBITMAP LoadResourceBitmap(HINSTANCE hInstance, LPSTR lpString,
                           HPALETTE FAR* lphPalette)
{
    HRSRC  hRsrc;
    HGLOBAL hGlobal;
    HBITMAP hBitmapFinal = NULL;
    LPBITMAPINFOHEADER  lpbi;
    HDC hdc;
    int iNumColors;

    if (hRsrc = FindResource(hInstance, lpString, RT_BITMAP))
       {
       hGlobal = LoadResource(hInstance, hRsrc);
       lpbi = (LPBITMAPINFOHEADER)LockResource(hGlobal);

       hdc = GetDC(NULL);
       *lphPalette =  CreateDIBPalette ((LPBITMAPINFO)lpbi, &iNumColors);
       if (*lphPalette)
          {
          SelectPalette(hdc,*lphPalette,FALSE);
          RealizePalette(hdc);
          }

       hBitmapFinal = CreateDIBitmap(hdc,
                   (LPBITMAPINFOHEADER)lpbi,
                   (LONG)CBM_INIT,
                   (LPSTR)lpbi + lpbi->biSize + iNumColors *
sizeof(RGBQUAD),

                   (LPBITMAPINFO)lpbi,
                   DIB_RGB_COLORS );

       ReleaseDC(NULL,hdc);
       UnlockResource(hGlobal);
       FreeResource(hGlobal);
       }
    return (hBitmapFinal);
}

HPALETTE CreateDIBPalette (LPBITMAPINFO lpbmi, LPINT lpiNumColors)
{
```

```
   LPBITMAPINFOHEADER  lpbi;
   LPLOGPALETTE     lpPal;
   HANDLE           hLogPal;
   HPALETTE         hPal = NULL;
   int              i;

   lpbi = (LPBITMAPINFOHEADER)lpbmi;
   if (lpbi->biBitCount <= 8)
       *lpiNumColors = (1 << lpbi->biBitCount);
   else
       *lpiNumColors = 0;  // No palette needed for 24 BPP DIB

   if (*lpiNumColors)
      {
      hLogPal = GlobalAlloc (GHND, sizeof (LOGPALETTE) +
                             sizeof (PALETTEENTRY) * (*lpiNumColors));
      lpPal = (LPLOGPALETTE) GlobalLock (hLogPal);
      lpPal->palVersion    = 0x300;
      lpPal->palNumEntries = *lpiNumColors;

      for (i = 0;  i < *lpiNumColors;  i++)
         {
         lpPal->palPalEntry[i].peRed   = lpbmi->bmiColors[i].rgbRed;
         lpPal->palPalEntry[i].peGreen = lpbmi->bmiColors[i].rgbGreen;
         lpPal->palPalEntry[i].peBlue  = lpbmi->bmiColors[i].rgbBlue;
         lpPal->palPalEntry[i].peFlags = 0;
         }
      hPal = CreatePalette (lpPal);
      GlobalUnlock (hLogPal);
      GlobalFree   (hLogPal);
   }
   return hPal;
}
```

Here is an example of how you might use the above function to load a bitmap from a resource and display it using a logical palette:

```
   {
   HBITMAP hBitmap,hOldBitmap;
   HPALETTE hPalette;
   HDC hMemDC, hdc;
   BITMAP bm;

   hBitmap = LoadResourceBitmap(hInst,"test", &hPalette);
   GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
   hdc = GetDC(hWnd);
   hMemDC = CreateCompatibleDC(hdc);
   SelectPalette(hdc,hPalette,FALSE);
   RealizePalette(hdc);
   SelectPalette(hMemDC,hPalette,FALSE);
   RealizePalette(hMemDC);
   hOldBitmap = SelectObject(hMemDC,hBitmap);
   BitBlt(hdc,0,0,bm.bmWidth,bm.bmHeight,hMemDC,0,0,SRCCOPY);
   DeleteObject(SelectObject(hMemDC,hOldBitmap));
   DeleteDC(hMemDC);
```

```
   ReleaseDC(hWnd,hdc);
   DeleteObject(hPalette);
   }
```

Win32s

On Win32s, LoadResource does not return a true global memory handle. This causes CreateDIBitmap to fail. To work around this problem, use GlobalAlloc to create a global memory handle.

Here is what LoadResourceBitmap should be on Win32s:

```
HBITMAP LoadResourceBitmap(HINSTANCE hInstance, LPSTR lpString,
                           HPALETTE FAR* lphPalette)
{
    HRSRC  hRsrc;
    HGLOBAL hGlobal, hTemp;
    DWORD dwSize;
    HBITMAP hBitmapFinal = NULL;
    LPBITMAPINFOHEADER  lpbi;
    LPSTR lpRes, lpNew;
    HDC hdc;
    int iNumColors;

    if (hRsrc = FindResource(hInstance, lpString, RT_BITMAP))
    {
       hTemp = LoadResource(hInstance, hRsrc);
       dwSize = SizeofResource(hInstance, hRsrc);
       lpRes = LockResource(hTemp);

       hGlobal = GlobalAlloc(GHND, dwSize);
       lpNew = GlobalLock(hGlobal);
       memcpy(lpNew, lpRes, dwSize);
       UnlockResource(hTemp);
       FreeResource(hTemp);

       lpbi = (LPBITMAPINFOHEADER)lpNew;

       hdc = GetDC(NULL);
       *lphPalette =  CreateDIBPalette ((LPBITMAPINFO)lpbi, &iNumColors);
       if (*lphPalette)
       {
          SelectPalette(hdc,*lphPalette,FALSE);
          RealizePalette(hdc);
       }

       hBitmapFinal = CreateDIBitmap(hdc,
                 (LPBITMAPINFOHEADER)lpbi,
                 (LONG)CBM_INIT,
                 (LPSTR)lpbi + lpbi->biSize + iNumColors * sizeof(RGBQUAD),
                 (LPBITMAPINFO)lpbi,
                 DIB_RGB_COLORS );

       ReleaseDC(NULL,hdc);
```

```
        GlobalUnlock(hGlobal);
        GlobalFree(hGlobal);
    }
    return (hBitmapFinal);
}
```

REFERENCES
==========

For more information on DIB-related functions, please review the Microsoft Windows SDK sample DIBVIEW.

Additional reference words: 1.20 1.25a 1.30 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiBmp

Retrieving the Style String for a TrueType Font

PSS ID Number: Q84132

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Windows version 3.1 introduces the concept of a font style. In previous versions of Windows, a font could have bold, italic, underline, and strikeout properties, which were supported by corresponding members of the LOGFONT and TEXTMETRIC structures. Windows 3.1 continues to support these properties, however, it also supports the concept of a style name for TrueType fonts.

Windows use of style names can be demonstrated by the ChooseFont dialog box in the common dialog boxes dynamic-link library (COMMDLG.DLL). The ChooseFont dialog box contains two list boxes named Font and Font Style. The Font list box contains a list of all face names and the Font Style list box contains a list of font styles for the currently selected face. For example, if any non-TrueType font (such as MS Sans Serif) is selected, the following styles will appear in the style list box:

```
Regular
Bold
Italic
Bold Italic
```

TrueType fonts may have these or more elaborate styles. For example, the "Lucida Sans" face includes the following style names:

```
Regular
Italic
Demibold Roman
Demibold Italic
```

MORE INFORMATION

================

As part of the TrueType support, the GetOutlineTextMetrics function can be used to retrieve metric information for TrueType fonts, including the style name.

GetOutlineTextMetrics is prototyped as follows:

```
DWORD GetOutlineTextMetrics(HDC hdc, UINT cbData,
                            LPOUTLINETEXTMETRIC lpotm);
```

The hdc parameter identifies the device context. GetOutlineTextMetrics retrieves the metric information for the font currently selected into the specified device context. For GetOutlineTextMetrics to succeed, the font must be a TrueType font. The sample code given below shows how to synthesize the style name for a non-TrueType font.

The cbData parameter specifies the size, in bytes, of the buffer in which information is returned.

The lpotm parameter points to an OUTLINETEXTMETRIC structure. If this parameter is NULL, the function returns the size of the buffer required for the retrieved metric information.

The OUTLINETEXTMETRIC structure contains most of the font metric information provided with the TrueType format. The relative parts of the structure are listed below:

```
typedef struct tagOUTLINETEXTMETRIC {
  .
  .
  .
    PSTR    otmpFamilyName;
    PSTR    otmpFaceName;
    PSTR    otmpStyleName;
    PSTR    otmpFullName;
} OUTLINETEXTMETRIC;
```

While these four members of the OUTLINETEXTMETRIC structure are defined as near pointers to strings (PSTR), they are actually offsets into the structure from the beginning of the structure. Because the length of these strings is not defined, an application must allocate space for them above and beyond the space allocated for the OUTLINETEXTMETRIC structure itself. The sample code below demonstrates this. It also demonstrates using GetOutlineTextMetrics in an application that will also work with Windows 3.0.

```
#include <windows.h>
#include <windowsx.h>
.
.
.
HFONT                hFont;
LPOUTLINETEXTMETRIC  potm;
TEXTMETRIC           tm;
int                  cbBuffer;

hFont = CreateFont( ..... );

hFont = SelectObject(hDC, hFont);

/*
```

```
 * Call the GetTextMetrics function to determine whether or not the
 * font is a TrueType font.
 */
GetTextMetrics(hDC, &tm);

/*
* GetOutlineTextMetrics is a function implemented in Windows 3.1
* and later. Assume fWin30 was determined by calling GetVersion.
*/
if (!fWin30 && tm.tmPitchAndFamily & TMPF_TRUETYPE)
{
   WORD (WINAPI *lpfnGOTM)(HDC, UINT, LPOUTLINETEXTMETRIC);

   /*
    * GetOutlineTextMetrics is exported from
    * GDI.EXE at ordinal #308
    */
   lpfnGOTM = GetProcAddress(GetModuleHandle("GDI"),
                             MAKEINTRESOURCE(308));

   /*
    * Call GOTM with NULL to retrieve the size of the buffer.
    */
   cbBuffer = (*lpfnGOTM)(hDC, NULL, NULL);

   if (cbBuffer == 0)
   {
     /* GetOutlineTextMetrics failed! */
     hFont = SelectObject(hDC, hFont);
     DeleteObject(hFont);
     return FALSE;
   }

   /*
    * Allocate the memory for the OUTLINETEXTMETRIC structure plus
    * the strings.
    */
   potm = (LPOUTLINETEXTMETRIC)GlobalAllocPtr(GHND, cbBuffer);

   if (potm)
   {
     potm->otmSize = cbBuffer;

     /*
      * Call GOTM with the pointer to the buffer. It will
      * fill in the buffer.
      */
     if (!(*lpfnGOTM)(hDC, cbBuffer, potm))
     {
        /* GetOutlineTextMetrics failed! */
        hFont = SelectObject(hDC, hFont);
        DeleteObject(hFont);
        return FALSE;
     }
```

```
      /*
       * Do something useful with the string buffers. NOTE: To access
       * the string buffers, the otmp???Name members are used as
       * OFFSETS into the buffer. They *ARE NOT* pointers themselves.
       */
      OutputDebugString((LPSTR)potm + (UINT)potm->otmpFamilyName);
      OutputDebugString((LPSTR)potm + (UINT)potm->otmpFaceName);
      OutputDebugString((LPSTR)potm + (UINT)potm->otmpStyleName);
      OutputDebugString((LPSTR)potm + (UINT)potm->otmpFullName);

      /* Don't forget to free the memory! */
      GlobalFreePtr(potm);
    }
    else
    {
      /* GlobalAllocPtr failed */
      hFont = SelectObject(hDC, hFont);
      DeleteObject(hFont);
      return FALSE;
    }
}
else
{
    /*
     * It was not a TrueType font, or Windows 3.0 is running.
     */
    LOGFONT lf;
    char    szStyle[LF_FACESIZE];
    LPSTR   p;

    GetObject(hFont, sizeof(LOGFONT), &lf);

    /*
     * Fabricate a style string. Important note! The strings
     * "Italic", "Bold", and "Regular" are only valid in English. On
     * versions of Windows localized for other countries, these
     * strings will differ.
     */
    szStyle[0] = '\0';

    if (lf.lfWeight >= FW_BOLD)
      lstrcpy(szStyle, "Bold ");

    /*
     * If it's "Bold Italic," concatenate.
     */
     p = szStyle + lstrlen(szStyle);

     if (lf.lfItalic)
       lstrcpy(p, "Italic");

     if (!lstrlen(szStyle))
       lstrcpy(szStyle, "Regular");

    /*
```

```
       * szStyle now holds what is equivalent to the otmpStyleName
       * member.
       */
      OutputDebugString(szStyle);
  }

  hFont = SelectObject(hDC, hFont);
  DeleteObject(hFont);
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiTt

Retrieving the Text Color from the Font Common Dialog Box

PSS ID Number: Q86331

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The common dialog box library (COMMDLG.DLL) provides the ChooseFont() routine that provides a common interface (Font dialog box) to specify the attributes for a font in an application. When the user chooses the Apply button in the Font dialog box, the application can format selected text using the specified font. This button is also useful in an application that allows the user to select more than one font simultaneously.

The Font dialog box provides a common method to specify a number of font attributes, including color. An application can send the CB_GETITEMDATA message to the Color combo box to retrieve the currently selected color.

This article discusses the procedure required to obtain all the information about the currently specified font.

MORE INFORMATION
================

The Font dialog box includes an Apply button if the application includes the CF_APPLY value in the specification for the Flags member of the CHOOSEFONT data structure. The dialog box includes the Color combo box if the CF_EFFECTS value is also specified. The remainder of this article assumes that the application has specified both of these values.

To properly process input from the Apply button, an application must install a hook function. For more information on installing a hook function from an application, query on the following words in the Microsoft Knowledge Base:

```
   steps adding hook function
```

The following code illustrates one method to process input from the Apply button:

```
   case WM_COMMAND:
      switch (wParam)
      {
         case psh3:  // The Apply button
```

```
         // Retrieve the font information...
            SendMessage(hDlg, WM_CHOOSEFONT_GETLOGFONT, 0,
                        (LONG)(LPLOGFONT)&lfLogFont);

         // Perform any required processing
         // (create the specified font, for example)

         // Retrieve color information...
            iIndex = (int)SendDlgItemMessage(hDlg, cmb4,
                                             CB_GETCURSEL, 0, 0L);
            if (iIndex != CB_ERR)
            {
               dwRGB = SendDlgItemMessage(hDlg, cmb4, CB_GETITEMDATA,
                                          (WORD)iIndex, 0L);
               wRed = GetRValue(dwRGB);
               wGreen = GetGValue(dwRGB);
               wBlue = GetBValue(dwRGB);

               wsprintf(szBuffer, "RGB Value is %u %u %u\r\n", wRed,
                         wGreen, wBlue);

               OutputDebugString(szBuffer);
            }
            break;

         default:
            break;
      }
      break;
```

The color information is not required to create the font; however, this information is required to accurately display the font according to the user's specification.

In an application that does not use the Apply button, the rgbColors member of the CHOOSEFONT data structure contains the selected color. In this case, no special processing to retrieve the color is required.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Retrieving Time-Zone Information

PSS ID Number: Q115231

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

In Windows NT, version 3.1, the time-zone strings are compiled into a resource that is linked into the CONTROL.EXE file. For Windows NT, version 3.5 and later and Windows 95, the time-zone strings have been moved into the registry.

In Windows NT, the time-zone strings are located in the key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\
   Microsoft\
   Windows NT\
   CurrentVersion\
   Time Zones.
```

In Windows 95, the time-zone strings are located in the key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\
   Microsoft\
   Windows\
   CurrentVersion\
   Time Zones.
```

For each time zone, the registry key TZI is formatted as follows:

```
LONG       Bias;
LONG       StandardBias;
LONG       DaylightBias;
SYSTEMTIME StandardDate;
SYSTEMTIME DaylightDate;
```

You can use this information to fill out a TIME_ZONE_INFORMATION structure, which is used when calling SetTimeZoneInformation().

Additional reference words: 3.50 4.00
KBCategory: kbprg
KBSubcategory: BseMisc

Returning CBR_BLOCK from DDEML Transactions

PSS ID Number: Q102584

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

=======

DDEML servers are applications that provide data to client applications. For some servers, this data gathering may be a lengthy process, as when gathering data from sources such as serial ports or a network. DDEML allows a server application to process data asynchronously in these situations by returning CBR_BLOCK from the DDE callback function.

MORE INFORMATION

================

In DDEML-based applications, while transactions can be either synchronous or asynchronous, only DDEML client applications may choose to establish either type of transaction when requesting data from a server application. DDEML server applications do not distinguish between synchronous and asynchronous transactions.

Asynchronous transactions can be very useful when client applications know that the partner server application will take some time to gather data. This type of transaction frees up the client to do other things while waiting for a notification from the server of data availability.

Server applications have no way of determining whether the client application has requested data synchronously or asynchronously. Request transactions on the server's side are always synchronous. When a client requests data, the server's callback receives an XTYP_REQUEST transaction, where the expected return value is a data handle. If the server application has to wait for data from a serial port, for example, access to the CPU by other applications will be delayed, thereby freezing the system until data arrives.

There are a couple of ways one can enable the server to gather data in an asynchronous manner, thereby allowing it to yield to other applications on the system while it gathers data. One method is to use CBR_BLOCK; another is to change the request transaction to a one-time ADVISE loop.

Method 1

Given that DDEML callbacks are not re-entrant, and that DDEML expects a data handle as a return value from the XTYP_REQUEST transaction (and transactions of XCLASS_DATA class), the server application can block the callback momentarily. It can do this by returning a CBR_BLOCK value after posting itself a user-defined message.

This way, the server application can gather data in the background while DDEML queues up any further transactions. The server can start gathering data when its window procedure gets the user defined message that was posted by its DDE callback function.

When a server application returns CBR_BLOCK for a request transaction, DDEML disables the server's callback function. It also queues transactions that are sent by DDEML after its callback has been disabled. This feature gives the server an opportunity to gather data while allowing other applications to run in the system.

As soon as data becomes available, then the server application can call DdeEnableCallBack() to re-enable the server callback function. Once the callback is re-enabled, DDEML will resend the same request transaction to the server's callback and this time, because data is ready, the server application can return the appropriate data handle to the client.

Transactions that were queued up because of an earlier block are sent to the server's callback function in the order they were received by DDEML.

The pseudo code to implement method 1 might resemble the following:

```
BOOL gbGatheringData = TRUE;   // Defined GLOBAlly.
HDDEDATA ghData = NULL;

HDDEDATA CALLBACK DdeServerCallBack(...)
   {
       switch(txnType)
       {
       case XTYP_REQUEST:

          // If the server takes a long time to gather data...
          // for this topic/item pair, then
          // post a user-defined message to the server app's wndproc
       // and return CBR_BLOCK... DDEML will block the callback
       // and queue transactions.

       if(bGatheringData) {
         PostMessage(hSrvWnd, WM_GATHERDATA, .....) ;
         return CBR_BLOCK;
       }
       else              // Data is ready, send back handle.
         return ghData;

       default:
         return DDE_FNOTPROCESSED;
```

```
      }
    }

LRESULT CALLBACK SrvWndProc(...)
  {
    switch (wMessage)
       {
          case WM_GATHERDATA:

             while (bGatheringData)
               {
                  // Gather data here while yielding to others
                  // at the same time!
      if(!PeekMessage(..))
        bGatheringData = GoGetDataFromSource (&ghData);
                  else {
                    TranslateMessage() ;
                    DispatchMessage ();
                  }
                }
           DdeEnableCallback (idInst, ghConv, EC_ENABLEALL);
              break ;

           default:
             return DefWndProc();
         }
     }
```

Method 2

Advise transactions in DDEML (or DDE) are just a continuous request link. Changing the transaction from a REQUEST to a "one time only" ADVISE loop on the client side allows the server to gather data asynchronously.

The client application can start an ADVISE transaction from its side and when the server receives a XTYP_ADVSTART transaction, return TRUE so that an ADVISE link is established. Once the link is established, the server can start gathering data, and as soon as it becomes available, notify the client of its availability.

This can be done by calling DdePostAdvise(). The server can use PeekMessage() to gather data if the data gathering process is a lengthy one, so that other applications on the system will get a chance to run. Once the client receives the data from the server in its callback (in its XTYP_ADVDATA transaction), it can disconnect the the ADVISE link from the server by specifying an XTYP_ADVSTOP transaction.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Rich Edit Control Does Not Support Unicode

PSS ID Number: Q128558

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The Rich Edit control included with Windows 95 and Windows NT version 3.51 does not support Unicode.

For a Unicode application to use the Rich Edit control, it must convert any strings passed to the control to ASCII text. This includes strings used in the FINDTEXT structure and the TEXTRANGE structure.

Additional reference words: 3.50 4.00
KBCategory: kbui
KBSubcategory: UsrCtl

Right-justified Windows and Menus in Hebrew Win95

PSS ID Number: Q152137

The information in this article applies to:

- Microsoft Win32 Software Development Kit for
  - Microsoft Windows 95, version 4.0
  - Microsoft Windows NT, version 3.51

SUMMARY

=======

The written languages and cultures of the Middle East have a general right-to-left orientation while Latin-based written languages and cultures have a left-to-right orientation. This means that in a Hebrew application, lists, menus, sets of buttons, or anything else that can have an alignment should be designed with a right-to-left orientation.

MORE INFORMATION

================

Under Hebrew Windows 95, you can create overlapped windows, edit controls, list boxes, and so on, that have a right-to-left orientation by calling CreateWindowEx(WS_EX_RIGHT...). When the resulting window is opened in Hebrew Windows 95, if a caption bar exists for that window, it is right-aligned. If the resulting window is a control, the control would also have a right-aligned property. For example, edit controls would have cursors that start from the right end of the client area.

To right-align the menu takes a bit more work. The MENUEX resource and the MFT_RIGHTJUSTIFY flag will need to be used.

The syntax for MENUEX is:

```
menuID MENUEX
BEGIN
    [{[MENUITEM itemText [, [id] [, [type] [, state]]]] |
    [POPUP    itemText [, [id] [, [type] [, [state] [, helpID]]]]
    BEGIN
        popupBody
    END]} ...]
END
```

The following is an example of a right justified menu:

```
...
#include "winuser.h"
...

GENERIC MENUEX DISCARDABLE
BEGIN
    POPUP "&File", , MFT_RIGHTJUSTIFY
    BEGIN
        MENUITEM "&New",     IDM_NEW, MFS_GRAYED
```

```
            MENUITEM "&Open...", IDM_OPEN, MFS_GRAYED
            MENUITEM "&Save",    IDM_SAVE, MFS_GRAYED
            MENUITEM "E&xit",    IDM_EXIT
        END
        POPUP "&Edit", , MFT_RIGHTJUSTIFY
        BEGIN
            MENUITEM "Cu&t\tCtrl+X",   IDM_CUT, MFS_GRAYED
            MENUITEM "&Copy\tCtrl+C",  IDM_COPY,MFS_GRAYED
            MENUITEM "&Paste\tCtrl+V", IDM_PASTE, MFS_GRAYED
            MENUITEM "Paste &Link",    IDM_LINK, MFS_GRAYED
        END

    END
```

SetMenuItemInfo can also be used to set the MFT_RIGHTJUSTIFY flag for individual menu items.

Additional reference words: 3.51 4.00 ME Arabic
KBCategory: kbprg
KBSubcategory: wintldev
95 available. Arabic Windows 95 is still in beta.

Rotating a Bitmap by 90 Degrees

PSS ID Number: Q77127

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY
=======

There are no Windows functions that directly rotate bitmaps. All techniques for rotating bitmaps in Windows involve copying the rows from a source bitmap into the columns of a destination bitmap. The following contains code for rotating a bitmap using GetPixel() and SetPixel(), and contains an outline of code for rotating device independent bitmaps (DIB).

MORE INFORMATION
================

A device dependent bitmap (DDB) can be rotated using the GetPixel() and SetPixel() functions. To rotate the bitmap, use the following code:

```
HBITMAP Rotate90(HDC hDC, HBITMAP hSourceBitmap)
{
  HBITMAP hOldSourceBitmap, hOldDestBitmap, hDestBitmap;
  HDC hMemSrc, hMemDest;
  int height, width;
  int i, j;
  BITMAP iSrcBitmap;

  // Step 1: Create a memory DC for the source and destination bitmaps
  //         compatible with the device used.

  hMemSrc = CreateCompatibleDC(hDC);
  hMemDest= CreateCompatibleDC(hDC);


  // Step 2: Get the height and width of the source bitmap.

  GetObject(hSourceBitmap, sizeof(BITMAP), (LPSTR)&SrcBitmap);
  width = SrcBitmap.bmWidth;
  height = SrcBitmap.bmHeight;


  // Step 3: Select the source bitmap into the source DC. Create a
  //         destination bitmap, and select it into the destination DC.

  hOldSourceBitmap = SelectObject(hMemSrc, hSourceBitmap);
```

```
  hDestBitmap = CreateBitmap(height, width, SrcBitmap.bmPlanes,
                    SrcBitmap.bmBitsPixel, NULL);

  if (!hDestBitmap)
    return(hDestBitmap);

  hOldDestBitmap = SelectObject(hMemDest, hDestBitmap);


  // Step 4: Copy the pixels from the source to the destination.

  for (i = 0; i < width; ++i)
      for (j = 0; j < height; ++j)
          SetPixel(hMemDest, j, width - 1 - i,
                                          GetPixel(hMemSrc, i, j));


  // Step 5: Destroy the DCs.

  SelectObject(hMemSrc, hOldSourceBitmap);
  SelectObject(hMemDest, hOldDestBitmap);
  DeleteDC(hMemDest);
  DeleteDC(hMemSrc);


  // Step 6: Return the rotated bitmap.

  return(hDestBitmap);
}
```

If the bitmap is larger, using GetPixel() and SetPixel() may be too slow. If this is the case, there are two options:

1. If the contents of the bitmap do not change, create two versions of the bitmap, the normal version and one that is rotated by 90 degrees. Load the appropriate bitmap as required.

-or-

2. Find some way to manipulate the bits of the bitmap that is faster than using SetPixel() and GetPixel(). The best way to do this is to convert the bitmap to a device independent bitmap. The following four steps detail how to create the DIB and to perform the rotation:

   a. Use GetDIBits() to convert the bitmap to a device independent format. It is necessary to create a BITMAPINFO structure appropriate for the bitmap. This will write the bitmap as a series of scanlines. Each scanline is padded so that it is DWORD aligned.

   b. Allocate memory for the destination bitmap. This bitmap requires as many scanlines as the width of the source bitmap. Each scanline is as many pixels wide as the height of the source bitmap. Also, the scanlines must be DWORD aligned.

c. For each scanline in the source bitmap, copy the pixels to the appropriate column in the destination bitmap. NOTE: The format for each scanline depends upon the number of bits per pixel. See the BITMAPINFO documentation for a description of the possible formats.

d. Use SetDIBits() to copy the device independent bits into a device dependent bitmap. Another BITMAPINFO structure, appropriate for the destination device is required for this step.

The four steps of this method require much more work than is required if GetPixel() and SetPixel() are used; however, this method may be faster because it directly manipulates the bits in the bitmap.

Additional reference words: 3.00 3.10 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiBmp

RPC CALLBACK Attribute and Unsupported Protocol Sequences

PSS ID Number: Q131495

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SUMMARY

The CALLBACK attribute of RPC does not support connection-less protocol sequences.

MORE INFORMATION

If an RPC interface has a procedure with the CALLBACK attribute, it can only make use of connection-oriented protocol sequences. The following protocol sequences are not supported:

- ncadg_ip_udp
- ncadg_ipx

If an RPC client tries to call a remote procedure that in turn calls a procedure back on client, the client will be able make the initial call, but when the server tries to call the procedure on client, the RPC run time generates exception 1726: The remote procedure call failed.

Additional reference words: 3.50
KBCategory: kbnetwork kbnetwork
KBSubcategory: NtwkRpc

RPC Can Use Multiple Protocols

PSS ID Number: Q100009

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, and 4.0

Microsoft Remote Procedure Call (RPC) does not rely on a single protocol. An RPC server can be written to use all available protocols on the server, and a client can be written in the same manner. Thus, the server or client does not have to know which protocols it supports explicitly.

RPC protocols supported by Windows 3.1 and 3.5 are:

ncacn_ip_tcp	(TCP/IP)
ncacn_nb_nb	(NetBIOS over NetBEUI)
ncacn_nb_tcp	(NetBIOS over TCP)
ncacn_np	(Named Pipes)
ncalrpc	(LPC)

RPC protocols supported by Windows 3.5 only are:

ncadg_ipx	(Datagram - IPX)
ncacn_spx	(SPX)
ncadg_ip_udp	(Datagram - UDP)
ncacn_nb_ipx	(Netbios over IPX)

RPC protocols supports by Windows 95 are:

- ncacan_ip_tcp
- ncacn_nb_nb
- ncacn_np
- ncacn_spx
- ncalrpc

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork
KBSubcategory: NtwkRpc

RpcNsxxx() APIs Not Supported by Windows NT Locator

PSS ID Number: Q104318

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

The RpsNSxxx group and profile name service application programming interfaces (APIs), such as RpcNsProfileEltAdd(), are supported by our RPC run time; however, they are not supported by the Windows NT Locator, which is the default RPC name service provider. If you attempt to make a call to one of these APIs, an error 1764, "request not supported," will be returned. Because the RpcNSxxx APIs are supported in the run time, name service providers other than the Locator, such as the DCE CDS, can be accessed.

Additional reference words: 3.10 3.50
KBCategory: kbnetwork
KBSubcategory: NtwkRpc

Running a Windows-Based Application in its Own VDM

PSS ID Number: Q115235

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.5

SUMMARY
=======

The Windows NT, version 3.1, Windows on Win32 (WOW) supports running all 16-bit Windows-based applications in one virtual machine (VM). These applications share an address space, just as they do on Windows. The Windows NT, version 3.5, WOW supports running a Windows-based application in its own VM, which gives the application its own address space.

In order to programmatically start a Windows-based application in its own VM, start the application with CreateProcess() and then specify the flag CREATE_SEPARATE_WOW_VDM.

To specify a Win16 application started from the command prompt to run in its own address space, use the following syntax:

```
start /SEPARATE <filename>
```

To specify a Windows-based application started from the Program Manager to run in its own address space, check the following item in the Program Item Properties for the application or in the Run dialog box, which can be selected from the Program Manager's File menu:

```
Run in Separate Memory Space
```

NOTE: This option is not the default, nor is there any way to make it the default.

MORE INFORMATION
================

Allowing a Windows-based application to run in a separate address space provides for more robust operation, because the application is isolated from other Windows-based applications. However, the downside is twofold:

- WOW VMs require approximately 2500K of private memory on x86 machines.

- There is no shared memory between WOW VMs. Therefore, Windows-based applications that rely on shared memory cannot be run in separate VMs. As an alternative, use DDE or OLE, because they can be used by an application in one VM to communicate with an application in another VM.

Additional reference words: 3.50
KBCategory: kbprg

KBSubcategory: Subsys

Running Bound Applications Under Windows NT

PSS ID Number: Q90913

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY

Bound applications are designed and built so they can be run under either OS/2 or MS-DOS. The OS/2 subsystem is not available on MIPS; therefore, bound applications will not run as OS/2 applications on MIPS.

When a bound application is run under Windows NT on an 80x86 system, the application will automatically run under the OS/2 subsystem if it is available.

MORE INFORMATION

The OS/2 subsystem is available by default on an 80x86 system. To force bound applications to run as an MS-DOS-based application, you can disable the OS/2 subsystem, but this is not recommended. Instead use the FORCEDOS facility. Type "FORCEDOS /?" at the command line to get help on FORCEDOS. It is not advised that you disable the OS/2 subsystem unless there is a very specific need that FORCEDOS does not address.

To disable the OS/2 subsystem using RegEdit:

   Go to HKEY_LOCAL_MACHINE
   Go to SYSTEM
   Go to Current Control Set
   Go to Control
   Go to Session Manager
   Go to SubSystems

   Modify 'Optional: REG_MULTI_SZ OS/2 Posix'. Specifically, remove OS/2 and reboot.

Once this is done, bound applications will run as MS-DOS-based applications. Running an OS/2 application results in the following message:

   Cannot connect to OS/2 subsystem

WARNING: REGEDT32 is a very powerful utility that facilitates directly changing the Registry database. Using REGEDT32 incorrectly can cause serious problems, including hard disk corruption. It may be necessary to reinstall the software to correct some problems. Microsoft does not support changes made with REGEDT32. Use this tool AT YOUR OWN RISK.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: SubSys

RW2002 Error "Cannot Reuse String Constants" in RC.EXE

PSS ID Number: Q21569

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

SUMMARY

The error "Cannot Reuse String Constants" will be returned by the Resource Compiler if you have used the same ID value to define two different string constants.

For example, the following error is returned when compiling the resource file:

```
Cannot Reuse String Constants
```

The file MY.RC may contain the following lines:

MORE INFORMATION

The following sample code can be used to demonstrate the problem.

Sample Code

```
StringTable
begin
   1, "one"
   2, "two"
   3, "three"
   1, "four"
end
```

Note that "one" and "four" have the same value. This error may be less noticeable if you are using both decimal and hexadecimal notation in your RC file. In the following example, 0x010 and 16 have the same value and generate the error:

```
 0x010, "hex 10"
 10, "ten"
 11, "eleven"
 15, "fifteen"
 16, "sixteen"
```

Additional reference words: 3.00 3.10 3.50 4.00 95 RW2002 RC2151
KBCategory: kbtool
KBSubcategory: TlsRc

SAMPLE: 16 and 32 Bits-Per-Pel Bitmap Formats

PSS ID Number: Q94326

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Windows NT supports the same bitmap formats as Microsoft Windows version 3.1, but includes two new formats: 16 and 32 bits-per-pel.

SEEDIB.EXE is a file in the Microsoft Software Library that contains the source code to an application that demonstrates how to load, display, and save 1, 4, 8, 16, 24, and 32-bits-per-pixel DIB formats. In addition, it demonstrates a simple method of creating an optimized palette for displaying DIBSs with more than 8-bits-per-pixel on 8-bits-per-pixel devices.

NOTE: In order to minimize color loss, SeeDIB uses CreateDIBSection() to do conversions between uncompressed DIBs which have more than 8-bits-per-pixel. This function is not available on Windows NT 3.1.

You can download SEEDIB.EXE from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download SEEDIB.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get SEEDIB.EXE

MORE INFORMATION

For DIBs (device independent bitmaps), the 16 and 32-bit formats contain three DWORD masks in the bmiColors member of the BITMAPINFO structure. These masks specify which bits in the pel correspond to which color.

The three masks must have contiguous bits, and their order is assumed to be R, G, B (high bits to low bits). The order of the three masks in the color table must also be first red, then green, then blue (RGB). In this manner, the programmer can specify a mask indicating how many shades of each RGB color will be available for bitmaps created with CreateDIBitmap(). For 16-bits-per-pixel DIBs, CreateDIBitmap() defaults to the RGB555 format. For 32-bits-per-pixel DIBs, CreateDIBitmap() defaults to an RGB888 format.

NOTE: The DIB engine in Windows 95 supports only RGB555 and RGB565 for 16-bit DIBs and only RGB888 for 32-bit DIBs.

Example

The RGB555 format masks would look like:

```
   0x00007C00  red   (0000 0000 0000 0000 0111 1100 0000 0000)
   0x000003E0  green (0000 0000 0000 0000 0000 0011 1110 0000)
   0x0000001F  blue  (0000 0000 0000 0000 0000 0000 0001 1111)
```

NOTE: For 16 bits-per-pel, the upper half of the DWORDs are always zeroed.

The RGB888 format masks would look like:

```
   0x00FF0000  red   (0000 0000 1111 1111 0000 0000 0000 0000)
   0x0000FF00  green (0000 0000 0000 0000 1111 1111 0000 0000)
   0x000000FF  blue  (0000 0000 0000 0000 0000 0000 1111 1111)
```

Usage

When using 16 and 32-bit formats, there are also certain fields of the BITMAPINFOHEADER structure that must be set to the correct values:

1. The biCompression member must be set to either BI_RGB or BI_BITFIELDS. Using BI-RGB indicates that no bit masks are included in the color table and that the default (RGB555 for 16bpp and RGB888 for 32bpp) format is implied. Using BI_BITFIELDS indicates that there are masks (bit fields) specified in the color table.

2. As with 24-bits-per-pixel formats, the biClrUsed member specifies the size of the color table used to optimize performance of Windows color palettes. If the biCompression is set to BI_BITFIELDS, then the optimal color palette starts immediately following the three DWORD masks. Note that an optimal color palette is optional and many applications will ignore it.

A technical note (VFW.ZIP) related to this subject from the Microsoft Multimedia group is also available.

The technical note is part of the Video for Windows technical notes and describes how to create a display driver that supports these new DIB formats, which are used by Video for Windows. The technical note also includes definitions of installable image Codecs.

Windows 95

In Windows 95, if the BI_BITFIELDS flag is set, then a color mask must be specified and it must be one of the following:

Resolution Bits per color Color Mask

16bpp	5,5,5	0x00007c00	0x000003e0	0x0000001f
16bpp	5,6,5	0x0000f800	0x000007e0	0x0000001f
32bpp	8,8,8	0x00ff0000	0x0000ff00	0x000000ff

User-defined color masks are not available in Windows 95.

Additional reference words: 3.10 3.50 4.00 bpp bmp
KBCategory: kbgraphic kbfile
KBSubcategory: GdiBmp

SAMPLE: Adding TrueType, Raster, or Vector Fonts to System

PSS ID Number: Q130459

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

FONTINST is a sample application in the Microsoft Software Library that demonstrates how to programmatically add a TrueType, raster, or vector font to the system.

When working with a TrueType font file (.TTF file), FONTINST creates a .FOT file and moves the specified .TTF file to the Windows system directory. For a raster or vector font file (.FON file), FONTINST moves the .FON file to the Windows system directory. After the font file has been moved to the system directory, FONTINST adds the font to the system by using the AddFontResource() API. Then it adds font information to the [fonts] section of the WIN.INI file so that the font is automatically loaded every time Windows starts. For example, the following line is added to the WIN.INI file when FONTINST adds the ARIAL.TTF file to the system:

```
Arial (TrueType)=ARIAL.FOT
```

FONTINST also demonstrates how to retrieve the facename of a font given a .TTF or .FON file. In the case of a TrueType font, FONTINST opens up the file and reads the naming table of the .TTF file. FONTINST also shows how to read the FONTINFO structure (as described on pages 49-50 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 4: Resources") of a .FON file. The facename of a font in a .FON file can also be found in this manner.

After a font is installed by FONTINST, information about the font is displayed in the window. Information such as the TEXTMETRIC structure and font type, as well as sample font text, is displayed. Information added to the WIN.INI file is also displayed in this window.

Download FONTINST.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download FONTINST.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get FONTINST.EXE

Additional reference words: 3.10 3.50 4.00 95 softlib
KBCategory: kbgraphic kbfile
KBSubcategory: GdiFnt GdiFntCreate

SAMPLE: AngleArc in Windows 3.1, Win32s, and Windows 95

PSS ID Number: Q125693

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SUMMARY

In Windows version 3.1, Win32s, and Windows 95, you may find it useful to get the functionality provided by the Win32 API AngleArc(). AngleArc() is only supported on Windows NT.

MORE INFORMATION

The AngleArc() function draws a line segment and an arc. The line segment is drawn from the current position to the beginning of the arc. The arc is drawn along the perimeter of a circle with the given radius and center. The length of the arc is defined by the given start and sweep angles. The starting point of the sweep is determined by measuring counterclockwise from the x-axis of the circle by the number of degrees in the start angle. The ending point is similarly located by measuring counterclockwise from the starting point by the number of degrees in the sweep angle.

The code below provides two possible ways of getting functionality similar to that of the AngleArc() function. While both of these methods will work on any Windows platform, the second (AngleArc2) will be substantially faster due to the fact that it uses the Arc() function to draw the sweep rather than calculating each of the segments on the perimeter of the arc.

NOTE: One limitation of the second method is that if the sweep angle is greater than 360 degrees, the arc will not be swept multiple times. In most cases this will not be a problem but in certain cases (constructing paths, for example) this can be a problem.

SAMPLE CODE #1

```
BOOL AngleArc1(HDC hdc, int X, int Y, DWORD dwRadius,
               float fStartDegrees, float fSweepDegrees)
{
  float fCurrentAngle;         // Current angle in radians
  float fStepAngle = 0.03f;  // The sweep increment value in radians
  float fStartRadians;       // Start angle in radians
  float fEndRadians;         // End angle in radians
```

```
  int ix, iy;                // Current point on arc
  float fTwoPi = 2.0f * 3.141592f;

  /* Get the starting and ending angle in radians */
  if (fSweepDegrees > 0.0f) {
    fStartRadians = ((fStartDegrees / 360.0f) * fTwoPi);
    fEndRadians = (((fStartDegrees + fSweepDegrees) / 360.0f) *
           fTwoPi);
  } else {
    fStartRadians = (((fStartDegrees + fSweepDegrees)  / 360.0f) *
          fTwoPi);
    fEndRadians =  ((fStartDegrees / 360.0f) * fTwoPi);
  }

  /* Calculate the starting point for the sweep via */
  /* polar -> cartesian conversion */
  ix = X + (int)((float)dwRadius * (float)cos(fStartRadians));
  iy = Y - (int)((float)dwRadius * (float)sin(fStartRadians));

  /* Draw a line to the starting point */
  LineTo(hdc, ix, iy);

  /* Calculate and draw the sweep */
  for (fCurrentAngle = fStartRadians;
       fCurrentAngle <= fEndRadians;
       fCurrentAngle += fStepAngle) {

    /* Calculate the current point in the sweep via */
    /* polar -> cartesian conversion */
    ix = X + (int)((float)dwRadius * (float)cos(fCurrentAngle));
    iy = Y - (int)((float)dwRadius * (float)sin(fCurrentAngle));

    /* Draw a line segment to current point */
    LineTo(hdc, ix, iy);
  }

  return TRUE;
}

SAMPLE CODE #2
==============

BOOL AngleArc2(HDC hdc, int X, int Y, DWORD dwRadius,
               float fStartDegrees, float fSweepDegrees)
{
  int iXStart, iYStart;  // End point of starting radial line
  int iXEnd, iYEnd;        // End point of ending radial line
  float fStartRadians;   // Start angle in radians
  float fEndRadians;     // End angle in radians
  BOOL bResult;            // Function result
  float fTwoPi = 2.0f * 3.141592f;

  /* Get the starting and ending angle in radians */
  if (fSweepDegrees > 0.0f) {
    fStartRadians = ((fStartDegrees / 360.0f) * fTwoPi);
```

```
    fEndRadians = (((fStartDegrees + fSweepDegrees) / 360.0f) *
          fTwoPi);
  } else {
    fStartRadians = (((fStartDegrees + fSweepDegrees)  / 360.0f) *
          fTwoPi);
    fEndRadians =  ((fStartDegrees / 360.0f) * fTwoPi);
  }

  /* Calculate a point on the starting radial line via */
  /* polar -> cartesian conversion */
  iXStart = X + (int)((float)dwRadius * (float)cos(fStartRadians));
  iYStart = Y - (int)((float)dwRadius * (float)sin(fStartRadians));

  /* Calculate a point on the ending radial line via */
  /* polar -> cartesian conversion */
  iXEnd = X + (int)((float)dwRadius * (float)cos(fEndRadians));
  iYEnd = Y - (int)((float)dwRadius * (float)sin(fEndRadians));

  /* Draw a line to the starting point */
  LineTo(hdc, iXStart, iYStart);

  /* Draw the arc */
  bResult = Arc(hdc, X - dwRadius, Y - dwRadius,
                     X + dwRadius, Y + dwRadius,
                     iXStart, iYStart,
                     iXEnd, iYEnd);

  /* Move to the ending point - Arc() wont do this and ArcTo() */
  /* wont work on Win32s or Win16 */
  MoveToEx(hdc, iXEnd, iYEnd, NULL);

  return bResult;
}
```

Additional reference words: 1.30 3.10 4.00
KBCategory: kbgraphic kbcode
KBSubcategory: GdiMisc

SAMPLE: Calling ExtDeviceMode/DeviceCapabilities in Win32s App

PSS ID Number: Q132239

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.25a

A Win32-based application running under Windows NT or Windows 95 can call DocumentProperties to change printer settings and DeviceCapabilities to query printer driver capabilities. But if the Win32-based appliction is running under Win32s, it has to load and call printer drivers directly by using calls such as LoadLibrary, GetProcAddress, ExtDeviceMode, and DeviceCapabilities.

Win32s handles these two printer driver functions specifically by creating a mapping thunk dynamically. The address returned to the Win32-based application by GetProcAddress is actually the address to the thunk that makes the 32-bit to 16-bit transition and calls the printer driver.

The DEVCAP sample shows:

- How to use DeviceCapabilities to query printer driver capabilities.
- How to use DocumentProperties under Windows NT or Windows 95 to change printing orientation.
- How to use ExtDeviceMode under Win32s to change printing orientation.
- How to work around two DeviceCapabilities bugs in the current version of Win32s (1.25.142) where DC_BINS and DC_PAPERS retrieve a list of DWORD items instead of a list of WORD items.

NOTE: Prior to version 1.25.142, Win32s contained a bug that didn't allow ExtDeviceMode to change printing orientation.

Download DEVCAP.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download DEVCAP.EXE
- Internet (anonymous FTP)
      ftp ftp.microsoft.com
      Change to the SOFTLIB\MSLFILES directory
      Get DEVCAP.EXE

Additional reference word: 3.50 4.00 95 softlib
KBCategory: kbprint kbfile

KBSubcategory: GdiPrn

SAMPLE: Changing Text Alignment in an Edit Control Dynamiclly

PSS ID Number: Q66942

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

A Microsoft Windows edit control aligns its contents to the left or right margins, or centers its contents, depending on the window style of the control. The control styles ES_LEFT, ES_CENTER, and ES_RIGHT specify left-, center-, and right-alignment, respectively.

Only multiline edit controls can be right-aligned or centered. Single-line edit controls are always left-aligned, regardless of the control style given.

Windows does not support altering the alignment style of a multiline edit control after it has been created. However, there are two methods that you can use to cause a multiline edit control in a dialog box to appear to change alignment. Note that in each of these methods, the dialog box that contains the control must be created with the DS_LOCALEDIT style.

MORE INFORMATION
================

The first method applies to all platforms. The second method does not apply to Windows 95. Under Windows 95, EM_SETHANDlE and EM_GETHANDLE are not supported. For more information, please see the following articles in the Microsoft Knowledge Base:

   ARTICLE-ID: Q130759
   TITLE   : EM_SETHANDLE and EM_GETHANDLE Messages Not Supported

Method 1

Create three controls: one left-aligned, one centered, and one right-aligned. Each has the same dimensions and position in the dialog box, but only one is initially made visible.

When the alignment is to change, call ShowWindow() to hide the visible control and to make one of the other controls visible.

To keep the text identical in all three controls, use the EM_GETHANDLE and EM_SETHANDLE messages to share the same memory among all three controls.

Method 2

Initially create a single control. When the text alignment is to change, retrieve location, size, and style bits for the existing edit control. Create a new control with the same size and in the same location, but change the style bits to reflect the new alignment.

Send the EM_GETHANDLE to each control to retrieve a handle to the memory that stores the contents. Send an EM_SETHANDLE to each control to exchange the memory used by each. Finally, destroy the original control.

There is a sample application named EDALIGN in the Microsoft Software Library that demonstrates each of these methods. Note, however, that this sample is a Windows 3.1 sample only.

Download EDALIGN.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

 - Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download EDALIGN.EXE

 - Internet (anonymous FTP)
      ftp ftp.microsoft.com
      Change to the \SOFTLIB\MSLFILES directory
      Get EDALIGN.EXE

Additional reference words: 3.00 3.10 3.50 4.00 95 EDALIGN.EXE
KBCategory: kbui kbfile
KBSubcategory: UsrCtl

SAMPLE: CHECKLCL Finds Corresponding Local Path for UNC Name

PSS ID Number: Q148391

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0
- Microsoft Windows NT Workstation versions 3.50 and 3.51
- Microsoft Windows NT Server version 3.5 and 3.51
- Microsoft Windows 95

SUMMARY

=======

UNC names specify network-wide distinct locations which are useful to provide configuration information for applications. However, when the UNC name refers to the local computer, it introduces some overhead as the requests are passed down the Redirector, looped back by the transport, and sent to the server service instead of going directly to the local disk.

The function provided in the library file in the CHECKLCL sample checks the name passed it against the local computer name and replaces the \\<computer>\<share> part with the local path to the share.

You can find CHECKLCL.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile CHECKLCL.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get CHECKLCL.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and Download CHECKLCL.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download CHECKLCL.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

MORE INFORMATION
================

Limitations or Caveats

- The function only resolves names in the Microsoft Windows Network. It
  doesn't work with FPNW.

- Because the changed names do not hit server service anymore, share
  security does not apply. But this is alright because the function, which
  is called to resolve the name, can only be called by Admin type accounts
  on Windows NT. On Windows 95, there is no local security anyway.

The CHECKLCL library has three callable functions: GlpGetLocalPathName, GlpFreeBuffer, and GlpGetBuffer.

The GlpGetLocalPathName Function

```
GlpGetLocalPathName( IN LPCTSTR pszInputName,
            IN OUT LPTSTR *ppszOutputName,
            IN OUT LPDWORD pdwOutputNameLength,
            OPTIONAL OUT LPDWORD pdwLanManError,
            OPTIONAL IN NETSHAREGETINFO_FPTR __stdcall
                        GlpNetShareGetInfoPtr
       );
```

To successfully execute the GlpGetLocalPathName function, you need to have administrative or server operator privileges on Windows NT. Also, the network should be running. The GlpGetLocalPathName function supports file, pipe, and mailslot share names. It also supports the use of path names longer than 260 characters (syntax \\?\ in CreateFile). For more information, please see the header file.

pszInputName
   The GlpGetLocalPathName function scans the name passed in pszInputName and tries to convert it to a local file path, if it's a UNC name referencing the local computer name.

ppszOutputName
   ppszOutputName contains the local path name if the return code is ERROR_GLP_SUCCESS. In any other case, the contents should be ignored. The GlpGetLocalPathName function will allocate memory if you pass a pointer to NULL. You should free the pointer using GlpFreeBuffer.

pdwOutputNameLength
   When you pass a pointer to memory in ppszOutputName, this parameter specifies the number of characters that can be stored there including the terminating NULL character.

   When the GlpGetLocalPathName function returns ERROR_GLP_SUCCESS, this pdwOutputNameLength variable contains the number of characters copied into the buffer.

If the function returns ERROR_GLP_INSUFFICIENT_BUFFER or ERROR_GLP_INSUFFICIENT_MEMORY, the pdwOutputNameLength variable contains the number of characters needed to store the converted name.

pdwLanManError
This optional parameter points to a DWORD that receives the error code returned by NetShareEnum if the return code is ERROR_GLP_LANMAN_ERROR. The value is not changed when NetShareEnum returns successfully. If you're not interested in this value, pass NULL to the function.

GlpNetShareGetInfoPtr
If not NULL, this function pointer will be called instead of NetShareEnum. This gives the caller the possibility to provide the local name or to call some other API. The parameters passed are for a call to NetShareEnum with information level 2. Remember that this function is UNICODE only.

The GlpFreeBuffer Function

```
GlpFreeBuffer (PVOID *ppszBuffer);
```

This function frees a pointer returned from GlpGetLocalPathName in the parameter ppszOutputName. The function sets the pointer to NULL after the memory is freed.

The GlpGetBuffer Function

```
GlpGetBuffer (PVOID *ppszBuffer, DWORD dwSizeNeeded);
```

Use this function if you pass a pointer to your custom NetShareGetInfo function to GlpGetLocalPathNameW to allocate memory for the bufptr output parameter of NetShareGetInfo. GlpGetLocalPathNameW will use its GlpFreeBuffer function to free the memory.

This function returns ERROR_GLP_SUCCESS on success and ERROR_GLP_INSUFFICIENT_MEMORY if there is not enough memory.

Additional reference words: 4.00 3.50
KBCategory: kbprg kbfile
KBSubcategory:

SAMPLE: Customizing the TOOLBAR Control

PSS ID Number: Q125683

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

The sample BARSDI demonstrates how to provide Customization features for the Toolbar Common Control. The Toolbar Common Control under Windows 95 provides Customization features that are useful when the user needs to change the toolbar control's buttons dynamically (add, delete, interchange, etc. buttons).

Download BARSDI.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download BARSDI.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get BARSDI.EXE

MORE INFORMATION

There are two ways the user can customize the toolbar:

First, the user can use the Drag Drop Customization process to delete or change the position of buttons on the toolbar. This method does not allow the user to add buttons to the toolbar dynamically.

The second method involves displaying the Customize dialog box through which the user can add, remove, interchange buttons on the toolbar.

To provide Customization, the toolbar control has to be created with the CCS_ADJUSTABLE style, and the parent of the toolbar control has to process a series of TBN_XXXX notifications. The BARSDI sample implements both methods of Customization.

Method 1: Drag Drop Customization

This method of toolbar customization allows the user to reposition or delete buttons on the toolbar. The user initiates this operation by holding down the SHIFT key and begins dragging a button. The toolbar

control handles all of the drag operations automatically, including the cursor changes.

To delete a button, the user has to release the drag operation outside the Toolbar control. The Toolbar control sends the TBN_QUERYDELETE message to its parent window. The parent window can return TRUE to allow the button to be deleted and FALSE to prevent the button from being deleted.

If the application wants to do custom dragging, it has to process the TBN_BEGINDRAG and TBN_ENDDRAG notifications itself and perform the drag/drop process, which involves more coding.

Method 2: Customization Dialog Box

This method of customization allows users to add buttons to the toolbar dynmacally in addition to deleting and rearranging buttons on the toolbar. For example, if the toolbar has N total buttons, and displays only 10 of those buttons initially, the bitmap that was used to create the toolbar, should contian all N buttons (where N > 10).

There are two ways in which the Toolbar control dislpays the customize dialog box. The user can bring up the Customization dialog box by double-clicking the left mouse button on the toolbar control or the application can send the TB_CUSTOMIZE message to the toolbar control.

The Customize dialog box displayed by the Toolbar control has two list boxes. One, on the left contains the list of N-10 Buttons that were not displayed on the initial toolbar, and the one on the right will have the currently displayed buttons on the toolbar. The toolbar control provides the add, remove and other features in the Customize dialog box.

Here is a code sample that shows how the Customization feature is implemented:

SAMPLE CODE
==========

```
// The initial set of toolbar buttons.

TBBUTTON tbButton[] =
{
    {0,   IDM_FILENEW,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {1,   IDM_FILEOPEN,    TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {2,   IDM_FILESAVE,    TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {3,   IDM_EDITCUT,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {0,   0,               TBSTATE_ENABLED, TBSTYLE_SEP,    0, 0},
    {4,   IDM_EDITCOPY,    TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {5,   IDM_EDITPASTE,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {6,   IDM_FILEPRINT,   TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {0,   0,               TBSTATE_ENABLED, TBSTYLE_SEP,    0, 0},
    {7,   IDM_ABOUT,       TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
};
```

```
// Buttons that can be added at a later stage.

TBBUTTON tbButtonNew[] =
{
    { 8,  IDM_ERASE,       TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    { 9,  IDM_PEN,         TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {10,  IDM_SELECT,      TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {11,  IDM_BRUSH,       TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {12,  IDM_AIRBRUSH,    TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {13,  IDM_FILL,        TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {14,  IDM_LINE,        TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {15,  IDM_EYEDROP,     TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {16,  IDM_ZOOM,        TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {17,  IDM_RECT,        TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {18,  IDM_FRAME,       TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {19,  IDM_OVAL,        TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
};

// The bitmap that is used to create the toolbar should have all
// tbButtonNew + tbButton buttons = 20 in this case.

// Use tbButtons array to create the initial toolbar control.

// Once the user starts to customize the toolbar, process the WM_NOTIFY
// message and the following notifications.
// The toolbar control sends a WM_NOTIFY message to the parent window
// during each process of the customization.

LRESULT OnMsgNotify(HWND hwnd, UINT uMessage, WPARAM wparam, LPARAM lparam)
{
    LPNMHDR         lpnmhdr;
    lpnmhdr =       (LPNMHDR)lparam;

// process the QUERYINSERT And QUERYDELETE notifications
// to allow the drag/drop operation to succeeed.
    if (lpnmhdr->code == TBN_QUERYINSERT)
        return TRUE;
    else if (lpnmhdr->code == TBN_QUERYDELETE)
        return TRUE;
    else if (lpnmhdr->code == TBN_GETBUTTONINFO)
// The user has brought up the customization dialog box,
// so provide the the control will button information to
// fill the listbox on the left side.
    {
        LPTBNOTIFY lpTbNotify = (LPTBNOTIFY)lparam;
        char  szBuffer [20];
        if (lpTbNotify->iItem < 12) // 20 == the total number of buttons
        {                           // tbButton and tbButtonNew
                                    // Since initially we displayed
                                    // 8 buttons
                  // send back information about the rest of
                  // 12 buttons that can be added the toolbar.

            lpTbNotify->tbButton = tbButtonNew[lpTbNotify->iItem];
```

```
            LoadString(hInst,
                       NEWBUTTONIDS + lpTbNotify->iItem, // string
                                                         //ID == command ID
                       szBuffer,
                       sizeof(szBuffer));

            lstrcpy (lpTbNotify->pszText, szBuffer);
            lpTbNotify->cchText = sizeof (szBuffer);
            return TRUE;
        }
        else
        return 0;
    }
}
```

Additional reference words: 4.00 BARSDI
KBCategory: kbui kbcode kbfile
KBSubcategory: UsrCtl

SAMPLE: DDRM.EXE: Mixing 2D DirectDraw Objects With Direct3D

PSS ID Number: Q152646

The information in this article applies to:

- Microsoft DirectX 2 Software Development Kit (SDK), for Windows 95

SUMMARY
=======

The DDRM sample demonstrates one method of combining 2D objects on a DirectDraw surface, such as a background, with Direct3D Retained Mode. It also demonstrates how to lock the primary surface's palette down and how to force Retained Mode to utilize what is in the palette and leave the entries in it unchanged. Since the palette will not change, the pixels of the 2D objects you create will not have to change. This sample renders to a full screen, 8 bit-per-pixel DirectDraw surface using double buffering.

You can find DDRM.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile DDRM.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get DDRM.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and download DDRM.EXE

- Microsoft Partner Network (MSPN)
  On MSPN Desktop, double-click the Software Library icon
  Search for DDRM.EXE
  Display results and download

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download DDRM.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

MORE INFORMATION
================

The DDRM sample creates a 640x480 primary surface with one back buffer. It also creates a 640x480 offscreen surface, on which it stores a 2D bitmap image. A palette is created for the 2D image and the palette is associated with both the back buffer and the front buffer. It is necessary to associate the palette with the back buffer because the back buffer will be rendered to by Retained Mode. To lock down the palette, you must set the peFlags member of the PALETTEENTRY structure for all palette entries to D3DPAL_READONLY. This will let Retained Mode know that it can use but not change the entries in the palette.

After obtaining a handle to the back buffer, a 16BPP z-buffer is created and attached to it. SetPalette() is called to attach the palette to the back buffer, and a destination color key for blit operations is created for the back buffer. The color key is palette index 255 (white), so when you blit to the back buffer, all pixels that are 255 (white) on the surface will be overwritten by the pixels on the surface you are blitting from. All other pixels will be preserved. Following is manner in which the destination color key is created:

```
DDCOLORKEY  ddck;
ddck.dwColorSpaceLowValue = 255;
ddck.dwColorSpaceHighValue = 255;
lpDDSBack->SetColorKey(DDCKEY_DESTBLT, &ddck);
```

After this is done, CreateDeviceFromSurface() is called to create a Retained Mode device for the back buffer.

When setting up the scene for the Retained Mode device, you must set the background color to 255 (white). When doing this, anything you blit to the back buffer will overwrite the background of the Retained Mode scene. Calling SetSceneBackground(D3DRGB(1,1,1)) will set up the background properly if white is the destination color key. It is important that only the background in the 3D scene contain white pixels. You should choose a background color (destination color key) you know will never be used on your 3D objects.

Following is the sequence of steps to take to update and render the display:

```
{
   HRESULT             ddrval;
   RECT                rcRect;
   DDBLTFX ddBltFx;

   rcRect.left = 0;
   rcRect.top = 0;
   rcRect.right = 640;
   rcRect.bottom = 480;

   // Clear the back buffer
   ZeroMemory(&ddBltFx,sizeof(DDBLTFX));
   ddBltFx.dwSize = sizeof(DDBLTFX);
```

```
      ddBltFx.dwFillColor = 255;
      lpDDSBack->Blt(NULL,NULL,NULL,DDBLT_COLORFILL | DDBLT_WAIT;
      ,&ddBltFx);

      // Update the 3D Retained Mode scene
      scene->Move(D3DVALUE(1.0));
      view->Clear();
      view->Render(scene);
      rmdev->Update();

      // Use DDBLTFAST_DESTCOLORKEY to blit the 2D bitmap image onto the
      // scene, only updating the white pixels
      while( (ddrval = lpDDSBack->BltFast( 0, 0, lpDDSOne, &rcRect,
                   DDBLTFAST_DESTCOLORKEY ) ) == DDERR_WASSTILLDRAWING );
      // Update the primary surface
      while(lpDDSPrimary->Flip( NULL,0 ) == DDERR_WASSTILLDRAWING);
   }
```

NOTE: In this version of Direct3D, you should not change the palette on your primary surface or your back buffer. Retained Mode will not account for the palette entry changes. Microsoft is aware of this problem. The DDRM sample demonstrates this problem by allowing you to change the palette while it is executing.

REFERENCES
==========

DirectDraw code from portions of the DDEX3 sample was used in parts of this sample.

Additional reference words: 4.00
KBCategory: kbprg kbgraphic kbfile
KBSubcategory: Direct3D

SAMPLE: Demonstration of OpenGL Material Property and Printing

PSS ID Number: Q136266

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The GLBMP sample illustrates how to define the material properties of the objects in the scene: the ambient, diffuse, and specular colors; the shininess; and the color of any emitted lights. This sample also demonstrates how to print an OpenGL image by writing the OpenGL image into a DIB section and printing the DIB section. The current version of Microsoft's implementation of OpenGL in Windows NT does not provide support for printing. To work around this current limitation, draw the OpenGL image into a memory bitmap, and then print the bitmap.

Download GLBMP.EXE a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download GLBMP.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get GLBMP.EXE

MORE INFORMATION

The GLBMP sample uses some of the new Windows 95 controls. So it won't run in Windows NT version 3.5 but will run in Windows NT version 3.51 and Windows 95 when OpenGL for Windows 95 is available.

Additional reference words: 4.00 glbmp.exe opengl
KBCategory: kbgraphic kbprint kbfile
KBSubcategory: GdiOpenGL GdiPrn

SAMPLE: Demonstration of the DrawEdge() Function

PSS ID Number: Q138320

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.5, 3.51, 4.0

The DrawEdge() function is useful for drawing user interface elements like the edges of buttons, sunken or raised areas, and other 3D border styles.

Drawedge is a sample application available for download that demonstrates how to use the DrawEdge() API function. The user can configure the various border and edge styles, and the application will display the results on the screen.

Download Drawedge.exe, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type mssupport
  Double-click the MS Software Library icon
  Find the appropriate product area
  Download Drawedge.exe

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download Drawedge.exe

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib\Mslfiles directory
  Get Drawedge.exe

Additional reference words: 4.00 3.50 UI GDI
KBCategory: kbui kbprg kbcode kbfile
KBSubcategory: GdiMisc

SAMPLE: Drawing Three-Dimensional Text in OpenGL Applications

PSS ID Number: Q131024

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51

SUMMARY

=======

GDI operations, such as TextOut, can be performed on an OpenGL window only if the window is single-buffered. The Windows NT implementation of OpenGL does not support GDI graphics in a double-buffered window. Therefore, you cannot use GDI functions to draw text in a double-buffered window, for example. To draw text in a double-buffered window, an application can use the wglUseFontBitmaps and wglUseFontOutlines functions to create display lists for characters in a font, and then draw the characters in the font with the glCallLists function.

The wglUseFontOutlines function is new to Windows NT 3.51 and can be used to draw 3-D characters of TrueType fonts. These characters can be rotated, scaled, transformed, and viewed like any other OpenGL 3-D image. This function is designed to work with TrueType fonts.

The GLFONT sample shows how to use the wglUseFontOutlines function to create display lists for characters in a TrueType font and how to draw, scale, and rotate the glyphs in the font by using glCallLists to draw the characters and other OpenGL functions to rotate and scale them. You need the Win32 SDK for Windows NT 3.51 to compile this sample, and you need to incorporate wglUseFontOutlines in your own application. You alos need Windows NT 3.51 to execute the application.

Download GLFONT.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download GLFONT.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory

  Get GLFONT.EXE

MORE INFORMATION

================

To specify the TrueType font for which you want wglUseFontOutlines to create display lists, you must first create the desired logical font by using CreateFont or CreateFontIndirect. Then, you must select the HFONT

created into a screen device context (HDC) with SelectObject, and send the HDC to the wglUseFontOutlines function. Each character is mapped in the x and y directions in the display lists. You need to specify the depth in the negative z direction in the extrusion parameter of wglUseFontOutlines.

You can also specify whether you want the 3-D glyphs to be created with line segments or polygons. To instruct wglUseFontOutlines to create the 3-D glyphs with lines segments, specify WGL_FONT_LINES in the format parameter. To create them with polygons, specify WGL_FONT_POLYGONS.

Here is an example showing how to create a set of display lists to draw the characters of the Arial TrueType font as a set of polygons:

```
     LOGFONT     lf;
     HFONT       hFont, hOldFont;
     GLYPHMETRICSFLOAT agmf[256];

     // An hDC and an hRC have already been created.
     wglMakeCurrent( hDC, hRC );

     // Let's create a TrueType font to display.
     memset(&lf,0,sizeof(LOGFONT));
     lf.lfHeight               =   -20 ;
     lf.lfWeight               =   FW_NORMAL ;
     lf.lfCharSet              =   ANSI_CHARSET ;
     lf.lfOutPrecision         =   OUT_DEFAULT_PRECIS ;
     lf.lfClipPrecision        =   CLIP_DEFAULT_PRECIS ;
     lf.lfQuality              =   DEFAULT_QUALITY ;
     lf.lfPitchAndFamily       =   FF_DONTCARE|DEFAULT_PITCH;
     lstrcpy (lf.lfFaceName, "Arial") ;

     hFont = CreateFontIndirect(&lf);
     hOldFont = SelectObject(hDC,hFont);

     // Create a set of display lists based on the TT font we selected
     if (!(wglUseFontOutlines(hDC, 0, 255, GLF_START_LIST, 0.0f, 0.15f,
        WGL_FONT_POLYGONS, agmf)))
          MessageBox(hWnd,"wglUseFontOutlines failed!","GLFont",MB_OK);

     DeleteObject(SelectObject(hDC,hOldFont));
     .  .  .  .
     .  .  .  .
     .  .  .  .
     .  .  .  .
```

To display these 3-D characters in a string, use the following code:

```
   // Display string with display lists created by wglUseFontOutlines()
   glListBase(GLF_START_LIST); // indicate start of display lists

   // Draw the characters
   glCallLists(6, GL_UNSIGNED_BYTE, "OpenGL");
```

Additional reference words: graphics
KBCategory: kbgraphic kbcode kbfile

KBSubcategory: GdiOpenGL

SAMPLE: Drawing to a Memory Bitmap for Faster Performance

PSS ID Number: Q130805

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.2

An application whose client area is a complex image can realize a performance benefit from drawing to a memory bitmap. The complex, time consuming drawing code need only be performed once - to initialize the offscreen bitmap. During the handling of the WM_PAINT message, the only work that needs to be done is a simple BitBlt from the memory bitmap to the screen.

Sample code demonstrating this technique is available in the Microsoft Software Library. The MemDC sample code draws a complex pattern on its client area. A menu option toggle allows the user to see the speed difference between using and not using the offscreen bitmap.

Download MEMDC.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download MEMDC.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get MEMDC.EXE

Additional reference words: 1.20 3.10 3.50 4.00 95 device context memory DC speed fast buffer
KBCategory: kbgraphic kbfile
KBSubcategory: GdiDc

SAMPLE: Fade a Bitmap Using Palette Animation

PSS ID Number: Q130804

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

PALFADE is a sample application available in the Microsoft Software Library. It demonstrates:

- How to use the AnimatePalette function to fade a bitmap to black.
- How to use the DIBAPI32.DLL library that can be built by the WINCAP32 sample that ships with the Microsoft Win32 SDK.

To perform palette animation, the sample creates a logical palette for a device-independent bitmap (DIB) with the PC_RESERVED flag set for each palette entry. PALFADE loads, displays, and animates both Windows-style and OS/2-style DIB files.

Download PALFADE.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download PALFADE.EXE
- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get PALFADE.EXE

MORE INFORMATION

Before performing palette animation on a logical palette entry, ensure that the palette entry has the PC_RESERVED flag set. To fade a bitmap drawn on a device context with a PC_RESERVED palette selected, you can lower the RGB values for each color in the palette in a loop until all colors are black.

The default system palette contains 20 static colors. These static colors take up the first ten and last ten colors of the system palette; these palette entries are not available for animation. If you try to fade a bitmap that has 256 unique colors by creating a 256-color palette with each palette entry set to PC_RESERVED, you are not guaranteed that every logical palette entry will map to an entry in the system palette that is available for palette animation.

One solution to this is to create a logical palette that contains only 236 colors. The PALFADE sample demonstrates one way to create an optimal palette of 236 colors given a device-independent bitmap with 256 colors in its color table.

Given a 256-color DIB, PALFADE traverses through every bit in the bitmap to find the least-used 20 colors in the color table. Then it creates a logical palette out of the 236 most-used colors. This ensures that all entries in the logical palette will animate.

This sample uses many of the DIB support functions included with the DIBAPI32.DLL library. It does not use the LoadDIB() function, because it was not written to handle OS/2-style DIB files. Instead, PALFADE implements the DIB-loading routines found in the Win32 SDK SHOWDIB sample.

NOTE: DIBAPI32.DLL is included with this sample.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic kbfile kbgraphic
KBSubcategory: GdiPal

SAMPLE: FASTBLT Implements Smooth Movement of a Bitmap

PSS ID Number: Q40959

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

There is a sample application called FASTBLT in the Microsoft Software Library that demonstrates how to implement the smooth movement of a bitmap around the screen. Basically, it sets up a pair of BitBlt() calls: one that erases the image and another that redisplays the image. The necessary ROP codes for BitBlt() that should be used are SRCCOPY and SRCINVERT.

Download FASTBLT.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download FASTBLT.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get FASTBLT.EXE

Additional reference words: 3.00 3.10 3.50 4.00 95 softlib FASTBLT.EXE
KBCategory: kbgraphic kbfile
KBSubcategory: GdiBmp

SAMPLE: FILEDRAG: How to Support File Drag Server Capabilities

PSS ID Number: Q139067

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SUMMARY

=======

Sample code is available that shows how an application can provide the File Drag drop server capabilities. It shows how to set up the data structures for CF_HDROP and ShellIDList formats. In this sample, the user can enable support for any combination of CF_HDROP and ShellIDList format and see the drag drop result. To obtain this sample:

Download FILEDRAG.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type mssupport
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Download FILEDRAG.EXE

- Microsoft Download Service (MSDL)
  - Dial (206) 936-6735 to connect to MSDL
  - Download FILEDRAG.EXE

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib\Mslfiles directory
  - Get FILEDRAG.EXE

MORE INFORMATION

================

In Windows 3.x, the only application that could act as a File Drag-Drop server was File Manager. Now the file drag-drop server capability is extended to all applications via OLE Drag Drop.

An OLE drag-drop server application that supports the CF_HDROP format can support file drag drop as does File Manager. The Windows system generates the appropriate WM_DROPFILES message for client applications that support File Drop, or the Windows system passes the IDataObject if the client application supports OLE Drop target for CF_HDROP format.

One of the new features of the shell is when you drop files on an icon on the desktop. The Shell opens the particular application with the drop file in it provided the application supports command line arguments where file can be passed as in this example:

```
   notepad.exe myfile.txt
```

This feature could also be provided by a file drag drop server that supports the ShellIDList format, in addition to the CF_HDROP format.

Additional reference words: 4.00
KBCategory: kbole kbfile kbhowto
KBSubcategory:

SAMPLE: GLEXT: Demo of GL_WIN_swap_hint & GL_EXT_vertex_array

PSS ID Number: Q139967

The information in this article applies to:

- Microsoft Win32 Software Development Kit for:

  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The GLEXT sample illustrates how to use the GL_WIN_swap_hint extension to speed up animation by reducing the amount of repainting between frames and how to use GL_EXT_vertex_array extension to provide fast rendering of multiple geometric primitives with one glDrawArraysEXT call. It also shows how to use glPixelZoom and glDrawPixels to display an OpenGL bitmap.

Download GLEXT.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type mssupport
  Double-click the MS Software Library icon
  Find the appropriate product area
  Download GLEXT.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download GLEXT.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib\Mslfiles directory
  Get GLEXT.EXE

MORE INFORMATION
================

Windows NT 3.51 and Windows 95 OpenGL supports OpenGL GL_WIN_swap_hint and GL_EXT_vertex_array extensions. To use GL_WIN_swap_hint extension, you need to use these functions:

```
glAddSwapHintRectWIN
SwapBuffers
glGetString
wglGetProcAddress
```

The glAddSwapHintRectWIN function lets you specify a set of rectangular areas that you want to copy when you call SwapBuffers function. When no rectangles are specified with glAddSwapHintRectWIN before calling

SwapBuffers, the entire frame buffer is swapped. If you only want to animate part of your scene, using glAddSwapHintRectWIN function will significantly speed up your animation. To check whether your implementation of OpenGL supports glAddSwapHintRectWIN, call glGetString(GL_EXTENSIONS). If it returns GL_WIN_swap_hint, glAddWsapHintRectWIN is supported. To obtain the address of the glAddSwapHintRectWIN function, call wglGetProcAddress.

To use GL_EXT_vertex_array extension, you need to use these functions:

- glDrawArraysEXT
- glGetString
- wglGetProcAddress

The glDrawArraysEXT function enables you to specify multiple geometric primitives to render. Instead of calling separate OpenGL functions to pass each individual vertex, normal, or color, you can specify separate arrays of vertexes, normals, and colors to define a sequence of primitives of the same kind, and just use one glDrawArraysEXT call to render them. To check whether your implementation of OpenGL supports glDrawArraysEXT, call glGetString(GL_EXTENSIONS). If it returns GL_EXT_vertex_array, glDrawArraysEXT is supported. To obtain the address of the glDrawArraysEXT function, call wglGetProcAddress.

You can use the following functions to define the data array for your primitives:

- glVertexPointerEXT
- glNormalPointerEXT
- glColorPointerEXT
- glTexCoordPointerEXT
- glEdgeFlagPointerEXT
- glArrayElementEXT
- glGetPointervEXT
- glIndexPointerEXT

See the SDK references for information on these functions.

Additional reference words: 3.51 4.00 glext.exe opengl extension
KBCategory: kbgraphic kbfile
KBSubcategory: GdiOpenGL

SAMPLE: GLTex Demos How to Use DIBs for Texture Mapping

PSS ID Number: Q148301

The information in this article applies to:

- Microsoft Win32 Software Development Kit for:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95

SUMMARY

The GLTex sample (GLTEXTUR.EXE) provides a demonstration of how to use a DIB (device-independent bitmap) as a texture-map for OpenGL by pasting a DIB (chosen by the user) onto all sides of a three-dimensional cube.

GLTex also allows you to modify various texture settings so that you can quickly and easily see the visual effect created.

You can find GLTEXTUR.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile GLTEXTUR.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get GLTEXTUR.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and Download GLTEXTUR.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download GLTEXTUR.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

MORE INFORMATION

OpenGL for Windows NT and Windows 95 support powerful texture-mapping capabilities. GLTex lets you easily experiment with many (but not all) of these capabilities by using a series of conversions to generate the texture map. It allows the user to modify various OpenGL settings.

To use a DIB as a texture in OpenGL, you must first convert the DIB into a format that is compatible with OpenGL. Because a 32-bpp DIB is the most closely related to an OpenGL formatted image, you can take the DIB that the user provides and regardless of its format copy it by using StretchDIBits onto a 32-bpp DIB Section. Therefore, from this point on, GLTex can assume that it is working with a 32-bpp DIB Section.

Note that because a 32-bpp DIB is used, you don't have to worry about the DWORD-alignment of DIBs. 32-bit DIBs are already DWORD-aligned. After ensuring that you have a 32-bpp DIB Section, you must convert that DIB Section into an OpenGL image. This involves converting the BGR format of the DIB into the RGBA format that OpenGL uses. This translation from an "any format DIB" into an OpenGL image is handled by GLTex's DIBtoGLImage function.

Now that you have an OpenGL image, you must scale it to a format that is 2^n x 2^n. This is required for OpenGL textures. (There is one exception to this rule involving texture borders, but GLTex does not use borders.) You might want to chose to always scale the image to be square because you're mapping onto cube faces. Then GLTex scales the image sides to the closest 2^n by using the utility function gluScaleImage. Finally, this scaled image is set as the texture-image by calling glTexImage2D. This scaling and setting of the texture is handled by GLTex's GLImagetoTexture function.

When the OpenGL scene is rendered, a simple three-dimensional cube is drawn with the converted and scaled texture mapped onto it. This mapping is taken care of by the glTexCoord function, which is called as the cube vertices are specified. This is handled by GLTex's BuildCube function.

Settings

Once you have loaded a DIB and see how it maps to the cube's sides, GLTex allows you to change several settings in order to see how these changes would effect the image. The following list gives modifications that GLTex allows you to make, a brief description of their effect, and which functions are used to change them:

- "Scene Distance from Viewer" allows you to push the scene farther away or bring it closer in. This is achieved with glTranslate.

- "Cube Rotation" allows you to modify the angle of rotation around the x, y, and z axes. This is achieved with glRotate for each axis.

- "Light Position" allows you to move the single light source used. Note that by changing the w coordinate, you can modify whether you are using a directional (w = 0.0) or positional (w = 1.0) light source. This is achieved with glLight(...GL_POSITION...).

- "Texture Mode" allows you to use two of the three texture modes

available with OpenGL. Decal mode essentially means that the texture is applied directly to the object without any calculation of material properties or lighting. Modulate mode blends the underlying object (which is effected by material properties and lighting) with the texture. This gives the effect of a lighted texture. The third possibility, Blending, doesn't make sense for a three-component (RGB) image. This is achieved with glTexEnv.

- "Perspective Hint" should be changed to Nicest if the texture appears to be projected incorrectly. You should try it anyway because it will allow you to see the difference. This is achieved with glHint.

- "Minification Filter" controls how a screen pixel should be mapped to a collection of texels (texture elements). This is achieved with glTexParameter(...GL_TEXTURE_MIN_FILTER...).

- "Magnification Filter" controls how a screen pixel should be mapped to a portion of a texel (texture element). This is achieved with glTexParameter(...GL_TEXTURE_MIN_FILTER...).

- "Texture Wrap S" and "Texture Wrap T" allow you to specify whether the image should repeat in the S and T directions (these can be thought of as the X and Y directions in the texture's world). This is achieved with glTexParameter(...GL_TEXTURE_<S or T>_WRAP...).

- "Texture Coordinates" specifies how the texture is mapped to the object. Because 0.0 is the default for the first parameter, one side of the texture is always mapped to the edge of the cube's face. This should be used in conjunction with Texture Wrap S and T for repeating the texture across the cube's faces. This is achieved with glTexCoord.

See the effects of these settings by experimenting with them in GLTex. When loading large DIBs with high-resolution, be patient, there is a lot of translation going on as described above, so depending on your system, this may take some time.

REFERENCES
==========

NOTE: Most of the following references can be found on the Microsoft Developer Network (MSDN) Developer Library CD-ROM. Some of these references are published by publishers independent of Microsoft; we make no warranty, implied or otherwise, regarding the reliability of these resources.

Code for this GLTex sample was borrowed from the ShowDIB sample for DIB-handling (Dib.c) and from Dale Rogerson's articles/samples for OpenGL palette creation (Glpalette.c).

For more information on the settings and OpenGL in general, please see the following resource:

- Neider, Jackie, Tom Davis, and Mason Woo. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1. Reading, MA: Addison-Wesley, 1993. ISBN 0-201-63274-8. (This book is also known as the "Red Book.")

For more information on DIBs and Texture Mapping in OpenGL, please see the following resources:

- Rogerson, Dale. "OpenGL V: Translating Windows DIBs." January 1995. (Development Library, Technical Articles)

- Rogerson, Dale. "OpenGL VII: Scratching the Surface of Texture Mapping." April 1995. (Development Library, Technical Articles)

For more information on DIBs in general, please see the following resources:

- Gery, Ron. "DIBs and Their Use.". March 1992. (Development Library, Technical Articles)

- Gery, Ron. "Using DIBs with Palettes." March 1992. (Development Library, Technical Articles)

Additional reference words: 3.51 4.00 image texture mapping
KBCategory: kbgraphic kbfile
KBSubcategory: GdiOpenGL

SAMPLE: Highlighting an Entire Row in a ListView Control

PSS ID Number: Q131788

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY

One of the limitations of the ListView common control is that when the control is in report view, the control only highlights the first column when a row is selected. To work around this limitation, you can create the ListView as an owner draw control (using the LVS_OWNERDRAWFIXED style) and perform all the painting yourself.

The ODLISTVW sample demonstrates how to create an owner draw ListView control that highlights an entire row.

Download ODLISTVW.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download ODLISTVW.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get ODLISTVW.EXE

MORE INFORMATION

One of the supported styles for the ListView control is LVS_OWNERDRAWFIXED, which allows the program to perform all the drawing of items in the ListView control. Whenever the ListView control needs to have a portion of the control repainted, it calculates which items are in that area and sends a WM_DRAWITEM message to its parent for each item. If any part of an item needs to be redrawn, the ListView sends the WM_DRAWITEM with the update rectangle set to the entire item.

The ListView control handles owner drawn items differently from other owner drawn controls. Previous owner drawn controls use the DRAWITEMSTRUCT's itemAction field to let the parent know if it needs to draw the item, change the selected state, or change the focus state. The ListView control always sends the WM_DRAWITEM message with the itemAction set to ODA_DRAWENTIRE. The parent needs to check the itemState to see if the focus or selection needs to be updated.

Additional reference words: 1.30 4.00 95 3.51 CListCtrl CListView list control
KBCategory: kbui kbfile kbwebcontent
KBSubcategory: UsrCtl

SAMPLE: How to Add to & Delete from the Windows 95 Start Menu

PSS ID Number: Q134333

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) version 4.0

SUMMARY
=======

The STRTMENU sample demonstrates how to add and delete program groups and items to and from the Windows 95 Start Menu without using DDE. This is accomplished by using combinations of shell functions and OLE.

MORE INFORMATION
================

Download STRTMENU.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

 - Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download STRTMENU.EXE

 - Internet (anonymous FTP)
      ftp ftp.microsoft.com
      Change to the SOFTLIB\MSLFILES directory
      Get STRTMENU.EXE

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbcode kbfile
KBSubcategory:

SAMPLE: How to Create & Play Enhanced Metafiles in Win32

PSS ID Number: Q145999

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 3.5

SUMMARY

=======

The Win32 SDK introduces a new type of metafile known as an enhanced metafile. These new metafiles address developer's need for device independence without requiring a separate code path, which was a requirement of the older style metafiles.

This article and the accompanying sample (ENMETA.EXE) show you how to properly create and play enhanced metafiles scaled or to original size. The sample also supports the clipboard and reads and writes Aldus placeable metafiles as well as regular 16-bit Windows metafiles.

MORE INFORMATION

================

You can find ENMETA.EXE, a self-extracting file, on these services:

- Microsoft's World Wide Web site on the Internet
  On the www.microsoft.com home page, click the Support icon
  Click Knowledge Base, and select the product
  Enter kbfile ENMETA.EXE, and click GO!
  Open the article, and click the button to download the file

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the Softlib/Mslfiles folder
  Get ENMETA.EXE

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type "mssupport" (without the quotation marks)
  Double-click the MS Software Library icon
  Find the appropriate product area
  Locate and Download ENMETA.EXE

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download ENMETA.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

Creating the Enhanced Metafile

An enhanced metafile is created with a call to CreateEnhMetaFile(). CreateEnhMetaFile() is declared as follows:

```
HDC CreateEnhMetaFile( HDC hdcRef, LPCTSTR lpFilename,
                       CONST RECT *lpRect, LPCTSTR lpDescription );
```

Following are the parameters to this call:

hdcRef is a reference DC and provides some crucial reference information for the construction of the metafile. The resolutions in both pixels and millimeters are taken from this device, and all font metrics are based on this DC. For the best possible reproduction on all output devices, it is a good idea to choose a high resolution device for the reference device. Otherwise, some grainyness may appear if the output device is of considerably higher resolution than the reference device.

lpFileName is the name of the file that will contain the metafile. If this parameter is NULL, the metafile is a memory metafile and is not stored on disk.

lpRect is a rectangle that specifies the dimensions in 0.01 mm units of the virtual metafile device. This rectangle is stored in the metafile and can be retrieved at play time to determine the desired real size of the output. Although no clipping is performed for this rectangle, some metafile players may presume that the image in the metafile fits in the rectangle.

lpDescription simply provides a way to store the name of the application that created the metafile as well as a description of the contents in the form of two NULL-terminated strings, terminated with an additional NULL as in this example:

```
   App\0Description\0\0
```

App is the name of the application that created the metafile and Description is a description of the image.

The Metafile Device

The metafile device has a real size defined by the lpRect parameter in 0.01 mm units. The number of device units on the metafile can be determined by using the pixel/mm ratio of the reference DC and the metafile device size given in the lpRect parameter. For example:

```
MetaPixelsX = MetaWidthMM * MetaPixels / (MetaMM * 100);
```

where MetaPixelsX = number of pixels on the X axis
MetaWidthMM = metafile width in 0.01mm units
MetaPixels = width in pixels of the reference device
MetaMM = width in millimeters of the reference device

A similar calculation can be used to determine the number of pixels in the

Y direction of the metafile device.

Note that although the metafile device has a real size, it does not clip output to that region. It is entirely possible to record drawing commands that have output outside of the metafile device surface.

Playing the Enhanced Metafile

The metafile device provides only half of the mapping from metafile space to target device space. The other half is provided by the RECT passed to the PlayEnhMetaFile() call. It specifies a play rectangle in which to play the metafile and is specified in logical coordinates on the target device context. The metafile device rectangle is mapped to this play rectangle. This provides the scalability of enhanced metafiles; adjusting the play rectangle adjusts the size of the output. No clipping is performed on the play rectangle.

So, neither the metafile device nor the play rectangle perform clipping. This means that if any drawing commands were recorded to occur outside the metafile device, they will be shown outside the play rectangle when the metafile is played. This mapping is illustrated in the Sample1.emf sample enhanced metafile.

If the goal is to play the metafile at its true size, the size of the original metafile device can be determined by a call to GetEnhMetaFileHeader(). This call fills in an ENHMETAHEADER structure, of which the rclFrame member specifies the metafile device rectangle in 0.01 mm units. That rectangle can be translated into logical units for the target DC and used as the lpRect parameter for PlayEnhMetaFile(). The PlayEnhMetaFileAtOriginalSize() function in the sample demonstrates this.

Mapping Modes

During the recording of the metafile, the mapping mode, window extents and origin, and viewport extents and origin combine to map logical units to device units on the metafile device. As with a normal DC, the window extents and origin define the logical space, while the viewport extents and origin are relative to the metafile device units described above.

For example: The following code creates an enhanced metafile that is dwInchesX wide by dwInchesY tall with dwDPI logical dots per inch (DPI):

```
HDC MyCreateEnhMetaFile( LPTSTR szFileName,  // Metafile filename
                         DWORD dwInchesX,    // Width in inches
                         DWORD dwInchesY,    // Height in inches
                         DWORD dwDPI )       // DPI (logical units)
{
    RECT   Rect = { 0, 0, 0, 0 };
    TCHAR   szDesc[] = "AppName\0Image Description\0\0";
    HDC      hMetaDC, hScreenDC;
    float   PixelsX, PixelsY, MMX, MMY;

    // dwInchesX x dwInchesY in .01mm units
```

```
    SetRect( &Rect, 0, 0, dwInchesX*2540, dwInchesY*2540 );

    // Get a Reference DC
    hScreenDC = GetDC( NULL );

    // Get the physical characteristics of the reference DC
    PixelsX = (float)GetDeviceCaps( hScreenDC, HORZRES );
    PixelsY = (float)GetDeviceCaps( hScreenDC, VERTRES );
    MMX = (float)GetDeviceCaps( hScreenDC, HORZSIZE );
    MMY = (float)GetDeviceCaps( hScreenDC, VERTSIZE );

    // Create the Metafile
    hMetaDC = CreateEnhMetaFile(hScreenDC, szFileName, &Rect, szDesc);
    // Release the reference DC
    ReleaseDC( NULL, hScreenDC );
    // Did you get a good metafile?
    if( hMetaDC == NULL )
        return NULL;

    // Anisotropic mapping mode
    SetMapMode( hMetaDC, MM_ANISOTROPIC );
    // Set the Windows extent
    SetWindowExtEx( hMetaDC, dwInchesX*dwDPI, dwInchesY*dwDPI, NULL );


    // Set the viewport extent to reflect
    // dwInchesX" x dwInchesY" in device units
    SetViewportExtEx( hMetaDC,
                      (int)((float)dwInchesX*25.4f*PixelsX/MMX),
                      (int)((float)dwInchesY*25.4f*PixelsY/MMY),
                      NULL );
    return hMetaDC;
}
```

Note that clipping is not performed by the window extents, viewport extents, or the metafile device. It is possible to draw beyond the window extents, which will map beyond the viewport extents. Those numbers provide only a ratio of logical units to metafile device units. Further, it is possible to have viewport extents that extend beyond the metafile surface. The drawing is not clipped to the metafile surface. This mapping is illustrated in the Sample2.emf sample enhanced metafile.

Palettes in Enhanced Metafiles

The SelectPalette() API can be used to record a palette into an enhanced metafile. That palette can then be retrieved at play-time via the GetEnhMetaFilePaletteEntries() API. This allows the player to faithfully reproduce palletized images and properly respond to palette messages. The GetEnhancedMetafilePalette() function in the sample demonstrates extracting a palette from an enhanced metafile.

Note that if multiple palettes were used in recording the metafile, the palette entries retrieved by GetEnhMetaFilePaletteEntries() will include all the entries from all the palettes. If this includes more entries than

the current display can reproduce, it is the player's responsibility to choose a subset of those colors from which to create the actual palette to be used during playback.

Additional reference words: 3.50
KBCategory: kbgraphic kbhowto kbcode kbfile
KBSubcategory: GdiMeta

SAMPLE: How to Create an Application Desktop Toolbar

PSS ID Number: Q134206

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The Windows 95 shell allows an application to register an application desktop toolbar that behaves with the same look and feel as the Windows 95 taskbar. The application desktop toolbar is always attached to one of the outside edges of the screen and can cause the size of the desktop to be reduced so that other applications do not overlap the application desktop toolbar.

The APPBAR sample code demonstrates how to implement an application desktop toolbar that is resizable, can attach to any side of the screen, and allows itself to be hidden in the same way the Windows 95 taskbar automatically hides.

Download APPBAR.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download APPBAR.EXE

- Internet (anonymous FTP)
      ftp ftp.microsoft.com
      Change to the SOFTLIB\MSLFILES directory
      Get APPBAR.EXE

MORE INFORMATION
================

The application desktop toolbar interface for Windows 95 provides an easy way for an application to attach windows to the edge of the screen so that they don't overlap the Windows 95 desktop or other applications. Being able to do this is especially useful in an application-launching program such as the Microsoft Office Manager (MOM) utility or in an application that needs to provide status information to the user as it runs in the background.

The application creates its window normally and then registers itself with the system as an application desktop toolbar (appbar). Once registered as an appbar, any time the window moves it must negotiate with the system for screen space by sending the requested rectangle to the system as part of an ABM_QUERYSETPOS message. The system then checks to see if any other appbars are using that space and adjusts the rectangle requested so as not to overlap. When the appbar is moved, the system resizes the desktop and moves

any currently running applications so they do not overlap the appbar.

REFERENCES
==========

For more information on the application desktop toolbar interface, please see the Chapter "Extending the Windows 95 Shell : Application Desktop Toolbars" in the "Programmer's Guide to Windows 95." The guide is available in the Win32 SDK Help file under "Guides."

Additional reference words: 4.00 Windows 95
KBCategory: kbui kbcode kbfile
KBSubcategory: UsrShell

SAMPLE: How to Draw Cubic Bezier Curves in Windows and Win32s

PSS ID Number: Q135058

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SUMMARY

You can get the functionality provided by the Win32 PolyBezier() function on the Windows 3.1 and Win32s platforms. The Bezier sample code shows by example how to draw cubic Bezier curves as well as curves with other degrees on both Windows version 3.1 and Win32s.

Download BEZIER.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download BEZIER.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get BEZIER.EXE

MORE INFORMATION

The PolyBezier() function draws cubic Bezier curves by using the endpoints and control points specified by the lppt parameter. The first curve is drawn from the first point to the fourth point by using the second and third points as control points. Each subsequent curve in the sequence needs exactly three more points: the ending point of the previous curve is used as the starting point, the next two points in the sequence are control points, and the third is the ending point.

Steps to Using the Sample Code

1. Compile the sample code using the .mak file that corresponds to the platform you are running (Bezier.mak for Win32s and Bezier16.mak for Win16).

2. Run the compiled program.

3. Use the left mouse button to position the endpoints and the control points in the client area. As soon as you've positioned enough points to draw a curve with the current degree (the default is cubic so you'll need four points), the curve will be drawn. Keep in mind that if you set the degree to a higher value, you'll have to position degree+1 points before the curve can be calculated and displayed.

Additional reference words: 1.30 3.10 4.00 Windows 95 Stones GDI
KBCategory: kbgraphic kbcode kbfile
KBSubcategory: GdiMisc

SAMPLE: How to Simulate Multiple-Selection TreeView Control

PSS ID Number: Q125587

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The TVWSTATE sample demonstrates how to simulate a multiple-selection TreeView control. The Windows 95 TreeView control does not support multiple selection. If you want a multiple-selection TreeView, you can use state images to simulate it in your application.

The TVWSTATE sample accomplishes this by using a checkbox type of state image to indicate that the item is selected or cleared (de-selected). These checkboxes will retain their state even if the TreeView loses focus.

Download TVWSTATE.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download TVWSTATE.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get TVWSTATE.EXE

MORE INFORMATION

The multiple-selection TreeView control simulation is implemented by setting the state image list of the TreeView control to an image list that contains the checked and unchecked checkbox bitmaps. This image list is set by using the TVM_SETIMAGELIST message with lParam == TVSIL_STATE (see InitImageList in TVWSTATE.C). A TreeView control can have two image lists, a normal image list and a state image list. In the TreeView, the display order from left to right is: the expansion button, the state image (if present), the normal image (if present), and then the item text.

When processing the WM_NOTIFY message where (LPNMHDR)lParam->code == NM_CLICK (see MsgNotifyTreeView in TVWSTATE.C), the code checks to see if the user clicked the left mouse button in the checkbox. If this is the case, the state image index of the item is retrieved, the index is toggled between the checked and unchecked image list items, and then the new index is saved.

The state image index identifies which member of the state image list should be displayed. The state image index is stored in bits 12-16 of the item state value. Either TVIS_STATEIMAGEMASK or TVIS_USERMASK can be used to mask off the lower bits. To access just the state image index, use a statement similar to this:

```
StateIndex = tvi.state & TVIS_STATEIMAGEMASK;
```

The INDEXTOSTATEIMAGEMASK macro offsets a value to the correct bits for the state image index. This is accomplished by shifting the given value left 12 places. If the desired state image index is 1, the state can be set using a statement similar to this:

```
tvi.state = INDEXTOSTATEIMAGEMASK(1);
```

This sample can also be modified to implement selection methods similar to those of an extended-selection listbox where the user uses the SHIFT key to select a range of items and/or the CTRL key to select or clear individual items.

Additional reference words: 3.51 4.00
KBCategory: kbui kbcode kbfile
KBSubcategory: UsrCtl

SAMPLE: How to Use File Associations

PSS ID Number: Q122787

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

Windows provides file associations so that an application can register the type of documents it supports. The benefit of doing this is that it allows the user to double-click or select a document in File Manager to edit or print it. File association is also supported by the ShellExecute() API. File associations also allow the user to open multiple documents with a single instance of the application via the File Manager.

ShellExecute() has even more benefit in Windows 95.

The sample FILEASSO.EXE demonstrates how to use file associations. Download FILEASSO.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download FILEASSO.EXE

- Internet (anonymous FTP)
      ftp ftp.microsoft.com
      Change to the SOFTLIB\MSLFILES directory
      Get FILEASSO.EXE

MORE INFORMATION

================

The following information applies to both File Open and File Print. For ease of reading, this article will discuss File Open to explain how File Associations work.

When the user double-clicks a document, the File Manager calls ShellExecute() with filename. ShellExecute() checks the Registration Database for an entry that associates that file extension with a particular application. If an entry exists and does not specify DDE commands, then ShellExecute() launches the application as specified in the registry. If the registry specifies to use DDE commands, ShellExecute() attempts to establish a DDE conversation with that application using the application topic. If an application responds to the DDE connections, ShellExecute() sends a DDE Execute command, as specified in the registry. It is up to the application to define the specifics on this conversation, particularly the

service and topic name to connect to, and also the correct DDE execute command syntax to use. However, if attempts to establish the conversation fail, ShellExecute() launches the application specified in the regiistry and tries to establish the DDE connection again.

There is one more option available when the application is not running, which seems to be appropriate for File Print. In this option, ShellExecute() sends a different Execute statement, the application needs to Open and Print the document. When the Printing is done, it exits.

There are two steps for an application to open multiple documents through single application instance via File association. As an example, assume MyApp is the application and AssocSupport is the topic. Most applications use MyApp as their application name and System as the topic.

1. When the application starts, register a DDE Server with the application name and topic (for example MyApp, and AssocSupport). The application also has to support DDE Execute Statements. The execute statement could be any format; at minimum, it should be:

   <Action> <fileName> <options>

   Here <Action> is anything specifying unique identification of the action, such as Open or Print. The <fileName> is the file that should be operated on. Finally, <options> can be any options that need to be passed on.

   A typical Execute Statement follows this format:

   [<Action>(<FileName>)]

   For example, Microsoft Word uses:

   [Open("%1")]

   The Application has to support the required functionality for executed statements.

2. File association can be done in Windows NT via File Manager or regedit.

Using the File Manager to Set File Associations

When associating a file type using the File Manager, choose Associate from the File menu. The Associate dialog presents the list of existing file associations. This dialog allows you to add a new file type (or file association), modify an existing file type, or delete an existing file type. The New File Type button allows the user to add an association for a new file extension. Here are the steps:

1. Add a File type name. For example, name it "Microsoft Word 6.0 Document."

2. Choose an action (Open or Print). For example, select the Uses DDE check box.

3. Add the directory path and application name. For example, enter
   WINWORD as the application.

4. Select the option Uses DDE.

5. Set the Application as the DDE Server Name.

6. Set the Topic as the DDE Server. For example enter System as the
   Topic.

7. Set the DDE Message <Action> <fileName> <options> to be the same as
   your application's Execute Statement. However the <fileName> and
   <options> should be replaced by whatever the command line arguments
   are. For example use:

      DDE Message : [FileOpen("%1")]

Using Regedit in Windows NT to Set File Associations

NOTE: Regedit is available only in Windows NT, not in Windows version 3.1.

The user can also associate files with an application by using regedit. From the Edit menu, choose Add File Type or Modify File Type (to modify an existing file type). A dialog similar to File Manager Associate dialog appears. Follow the same steps as described for File Manager. In Windows version 3.1, once you have defined a File Type via this method, go to the File Manager associate dialog and attach the file type to the extension.

Using a Program to Set File Assoications

You can also set the associations programmatically. This is useful when setting up your application on other machines. You would provide this functionality through your installation program. The first way to do this (the simplier method) is to use regedit to merge the changes from a file. The syntax for this is:

   regedit <filename>.reg

An example of a <filename>.reg is:

   REGEDIT
   HKEY_CLASSES_ROOT\.riy = FMA000_File_assoc
   HKEY_CLASSES_ROOT\FMA000_File_assoc = File_assoc
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\open\command = fileasso.EXE
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\open\ddeexec = [Open(%1)]
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\open\ddeexec\application
= Myserver
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\open\ddeexec\topic = system
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\print\command = fileasso.EXE
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\print\ddeexec = [Open(%1)]

HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\print\ddeexec\application =

```
MYServer
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\print\ddeexec\topic = System
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\print\ddeexec\ifexec =
[Test(%1)
]
```

In the program, you can also add keys to the registry by using the registry APIs. The developer needs to add the following keys to the registration database:

```
   // Your extensions.
   HKEY_CLASSES_ROOT\.riy = FMA000_File_assoc

   //File type name.
   HKEY_CLASSES_ROOT\FMA000_File_assoc = File_assoc

   // Command to execute when application is not running or dde is not
   // present and Open command is issued.
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\open\command = fileasso.EXE

   // DDE execute statement for Open.
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\open\ddeexec = [Open(%1)]

   // The server name your application responds to.
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\open\ddeexec\application =
   Myserver

   // Topic name your application responds to.
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\open\ddeexec\topic = system

   // Command to execute when application is not running or dde is not
   // present and print command is issued.
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\print\command = fileasso.EXE

   // DDE execute statement for Print.
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\print\ddeexec = [Open(%1)]

   // The server name your application responds to.
   HKEY_CLASSES_ROOT\FMA000_File_assoc\shell\print\ddeexec\application =
   MYServer

   // Topic name your application responds to.
   HKEY_CLASSES_ROOT\FMA000_File assoc\shell\print\ddeexec\topic = System

   // DDE execute statement for print if the application is not already
   // running. This gives the options for a an application to Run, Print
   // and Exit.
   HKEY_CLASSES_ROOT\FMA000_File assoc\shell\print\ddeexec\ifexec =
   [Test(%1)]
```

REFERENCES
==========

Windows SDK Programmers Reference, Volume 1, chapter 7, Shell Library or Books Online.

Window 3.1 SDK Help file, Registration Database, Shell Library Functions.

Win32 Programmers Reference, Volume 2, chapter 52, Registry and
Initialization Files or Books Online.

Win32 SDK Help file Registry and Initialization

File Manager Help File.

REGEDIT.HLP

REGEDT32.HLP

Additional reference words: 3.10 3.50
KBCategory: kbui kbfile
KBSubcategory: UsrMisc

SAMPLE: How to Use mciSendString() to Change an .AVI Palette

PSS ID Number: Q139746

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Video for Windows Development Kit (VfWDK) version 1.1

SUMMARY

=======

The SETPAL sample demonstrates how to use the "setvideo palette handle to" Media Control Interface (MCI) string to change the palette that will be used when an Audio-Video Interleaved (.avi) file is played.

Download SETPAL.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type mssupport
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Download SETPAL.EXE

- Microsoft Download Service (MSDL)
  - Dial (206) 936-6735 to connect to MSDL
  - Download SETPAL.EXE

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib\Mslfiles directory
  - Get SETPAL.EXE

The current video mode of the display must be a palettized video mode such as 256 colors for the technique to succeed because the "setvideo palette handle to" string is ignored in non-palettized video modes. Also, it is up to the video compressor/decompressor (codec) that decompresses the .avi file to handle the palette. Therefore, the exact behavior may depend on the codec in use. For example, the MS Video 1 codec is an 8-bit (256 color) palettized codec while the Indeo and Cinepak Codecs are 24-bit non-palettized Codecs, so their palette behavior may differ.

MORE INFORMATION

================

The basic steps to supply your own palette are as follows:

1. Fill in a LOGPALETTE structure containing the color information for your palette.

2. Create a logical palette using the CreatePalette() function.

3. Call mciSendString() (or mciSendCommand()) to supply that palette to
   the AVI file using a string such as:

   "setvideo <alias> palette handle to <palette handle>"

The remainder of this article supplies more information and code excerpts
from the SETPAL sample that perform these steps.

The SETPAL sample can be compiled as a 32-bit sample, called SETPAL32, or
as a 16-bit sample, called SETPAL16. As a result, it is compatible with
Windows 95, Windows NT, and Windows 3.1. Setpal32.mak and Setpal16.mak make
files are provided. SETPAL16 requires the Video for Windows 1.1 Development
Kit to build successfully.

SETPAL is a sample application that allows opening an .avi file via the
open common dialog box. Menu choices allow the .avi file to be opened,
played, stopped, or closed. MCI strings perform the underlying work. The
following MCI string is used to supply a palette for the .avi file:

   "setvideo <alias> palette handle to <palette handle>"

Excerpts from Setpal.c

```
// include files
#include <windows.h>  // required for all Windows applications
#include "windowsx.h" // for GlobalAllocPtr/GlobalFreePtr in
                      //   CreateSamplePalette
#include "mmsystem.h" // for the MCI calls

// global variables
static HPALETTE g_hPal = NULL;    // palette handle
.
.
.

// CreateSamplePalette() demonstrates how to fill in a LOGPALETTE
// structure and create a logical palette
VOID CreateSamplePalette(void)

{
    LPLOGPALETTE lpLogPal;
    int          i;
    int          nPalEntries = 236;   // number of entries in our palette
                                      // 256 are possible, but the system
                                      // reserves 20 of them

    lpLogPal = (LPLOGPALETTE) GlobalAllocPtr (GHND,
                sizeof (LOGPALETTE) + nPalEntries * sizeof (PALETTEENTRY));

    lpLogPal->palVersion    = 0x300;
    lpLogPal->palNumEntries = nPalEntries;

    for (i = nPalEntries; i > 0; i--)
```

```
    {
        // Fill in the red, green, and blue values for our palette.
        // This particular palette is a wash from green to black.
        lpLogPal->palPalEntry[i].peRed   = 0;
        lpLogPal->palPalEntry[i].peGreen = i;
        lpLogPal->palPalEntry[i].peBlue  = 0;

        // Create unique palette entries. This flag may change depending on
        // your purposes. See the Windows API documentation
        // concerning the PALETTEENTRY structure for more information.
        lpLogPal->palPalEntry[i].peFlags = PC_NOCOLLAPSE;
    }

    // create the logical palette
    g_hPal = CreatePalette (lpLogPal);

    // clean up
    GlobalFreePtr (lpLogPal);
}

// The ProcessAVICommands() function in SETPAL.C handles the open, set
// palette, play, and close for the AVI file. Once the AVI file has been
// opened, issue an mciSendString() such as the following to set the
// palette:

static char szAlias[10] = "paltest";  // the movie alias to use in
mciSendString
char szBuffer[128];                   // buffer to hold the MCI
string we built
.
.
.
   wsprintf(szBuffer, "setvideo %s palette handle to %d", (LPSTR)szAlias,
            g_hPal);
   mciSendString(szBuffer, NULL, 0, NULL);
```

Additional reference words: mciSendCommand MCI_SETVIDEO
MCI_DGV_SETVIDEO_ITEM MCI_DGV_SETVIDEO_PALHANDLE
KBCategory: kbmm kbprg kbfile kbhowto kbcode
KBSubcategory: MMVideo

SAMPLE: How to Use Paths to Create Text Effects

PSS ID Number: Q128091

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

This article defines the term "path" for the purposes of this article, and it explains how you can get sample code (provided in TEXTFX.EXE, a self-extracting file) that shows by example how to use paths to draw text at varying angles, orientations, and sizes. In addition, the sample code gives useful routines for displaying path data.

Download TEXTFX.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download TEXTFX.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get TEXTFX.EXE

MORE INFORMATION

A path is one or more figures (or shapes) that are filled, outlined, or both filled and outlined. Computer-aided design (CAD) applications use paths to create unique clipping regions, to draw outlines of irregular shapes, and to fill the interiors of irregular shapes.

A path is associated with a Device Context (DC) but unlike other objects associated with a DC, such as pens and brushes, a path has no default object.

To create a path, you first call BeginPath(). Then use the drawing functions in the table below to add to the path. Any drawing done using these functions is recorded as part of the path. When you finish building the path, call EndPath(). The new path can then be converted to a region by using PathToRegion(), selected as a clipping region for a device context by using SelectClipPath(), and rendered by using StrokePath() or FillPath(). In addition, as this sample illustrates, the path can be retrieved by using GetPath() and manipulated programmatically.

Functions supported in paths:

AngleArc	LineTo	Polyline
Arc	MoveToEx	PolylineTo
ArcTo	Pie	PolyPolygon
Chord	PolyBezier	PolyPolyline
CloseFigure	PolyBezierTo	Rectangle
Ellipse	PolyDraw	RoundRect
ExtTextOut	Polygon	TextOut

Functions not supported under Windows 95:

AngleArc
ArcTo
PolyDraw

Functions not supported in a path under Windows 95:

Arc
Chord
Ellipse
Pie
Rectangle
RoundRect

MORE INFORMATION
==========

The following path functions are used in the TextFX sample:

- BeginPath
- EndPath
- GetPath
- FillPath
- StrokePath

To use TextFX, run it, and then draw two lines into the client area (they don't have to be straight). The first line appears as blue and the second appears as red. These lines serve as guides for how the text will be rendered. After completing the second line, the text "This is a test" will be drawn so that it appears between the two guide lines.

To remap the text so that it appears between the two lines, TextFX first breaks down the guide lines (which are composed of line segments) into distinct adjacent points. The result is that the x,y position of each point in the lines is adjacent to its neighboring points x,y position.

Next, the text "This is a test" is drawn into a device context as a path. The points that make up the lines and curves in this path are then retrieved from the device context by using the GetPath() function.

To reposition the points in the path data, the code must establish a relationship between the relative position of the points in the original text and the position defined by the guide lines. To establish this relationship, the code calculates the x and y positions of each point in the path data relative to the overall extent of the text string. The

relative x position is used to calculate a corresponding point on each of the two the guide lines, while the relative y value is used as a weight to determine how far along on a line between the two guide line points the remapped position should be.

For example, if the point in the upper left corner of the "T" in the string "This is a test" is 2% of the total x extent of the string and 10% of the total y extent, then TextFX would find the point in each guide line that corresponds to 2% of the total number of points in that guide line. Then TextFX would reposition the point in the path data representing the upper left corner of the "T" so that it would be 10% of the way along an imaginary line extending from the point on the top guide line to the point on the bottom guide line.

Two different methods can be selected for drawing the remapped data, one draws just the outline of the characters, while the other fills in the characters.

To draw the outline of the characters, TextFX converts the remapped data back into a path and uses StrokePath() to display the outlines. To do the conversion, TextFX begins a new path, and then loops through the remapped data and uses the vertex types returned from GetPath() to determine how to draw the points. After drawing all the data, TextFX ends the path and calls StrokePath().

To draw the solid characters, instead of using StrokePath(), TextFX uses FillPath(). However, in order to get the interior areas of characters like "O", "A", "D", and so on, TextFX sets the ROP2 code to R2_MERGEPENNOT before calling FillPath(). This is done so that characters like "O" that consist of two separate polygons (one representing the outer perimeter and one representing the inner perimeter) will not be drawn as a solid blob. By drawing the polygons with the R2_MERGEPENNOT code, the code ensures that the second polygon will cancel the effects of the first in the area of the inner polygon.

REFERENCES
==========

For additional information on paths, please see the PATHS sample included with the Win32 SDK.

Additional reference words: 3.10 3.50 4.00 stones effects effect font fx
KBCategory: kbgraphic kbfile kbcode
KBSubcategory: GdiDrw

SAMPLE: How to Use the ToolTip Common Control

PSS ID Number: Q134209

The information in this article applies to:

 - Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0
 - Microsoft Win32s version 1.30

SUMMARY
=======

The TOOLTIPS sample code demonstrates how to use the ToolTip common control including how to use ToolTips in controls, rectangles (areas), and dialog boxes.

MORE INFORMATION
================

Download TOOLTIPS.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

 - Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download TOOLTIPS.EXE

 - Internet (anonymous FTP)
      ftp ftp.microsoft.com
      Change to the SOFTLIB\MSLFILES directory
      Get TOOLTIPS.EXE

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbfile
KBSubcategory:

SAMPLE: How to Use Three Different Styles of Property Sheets

PSS ID Number: Q132919

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0
- Microsoft Win32s version 1.30

SUMMARY

The PROPERTY sample code demonstrates how to use three different styles of property sheets: modal, modeless, and wizard.

Download PROPERTY.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download PROPERTY.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get PROPERTY.EXE

MORE INFORMATION

The main problem in dealing with a modeless property sheet is knowing when to destroy the property sheet. This is handled in the application's main message loop using the PSM_GETCURRENTPAGEHWND message (PropSheet_GetCurrentPageHwnd macro). PSM_GETCURRENTPAGEHWND returns NULL after the OK or Cancel buttons have been clicked (or the dialog box has been closed) and all of the pages have been notified. The application can then call DestroyWindow on the window handle that was returned from the PropertySheet call.

The sample also demonstrates how to use the PropSheetCallback function to modify the property sheet dialog box before or after it is created. The Wizard property sheet is modified to contain a system menu.

Additional reference words: 4.00 1.30 Windows 95
KBCategory: kbprg kbcode kbfile
KBSubcategory:

SAMPLE: Implementing Multiple Threads in an OpenGL Application

PSS ID Number: Q128122

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

It is possible to create multiple threads in an OpenGL application and have each thread call OpenGL functions to draw an image. You might want to do this when multiple objects need to be drawn at the same time or when you want to have certain threads perform the rendering of specific types of objects.

This article explains how to obtain GLTHREAD, a sample that demonstrates how to implement multiple threads in an OpenGL application. The main process default thread creates two threads that each draw a three-dimensional wave on the main window. The first thread draws a wave on the left side of the screen. The second thread draws a wave on the right side of the screen. Both objects are drawn simultaneously, demonstrating OpenGL's ability to handle multiple threads.

How to Get the GLTHREAD Sample

Download GLTHREAD.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download GLTHREAD.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get GLTHREAD.EXE

MORE INFORMATION

When implementing multiple threads in any type of application, it is important to have adequate communication between threads. In OpenGL, it is important for two threads to know what the other thread is doing. For example, it is common practice to clear the display window before drawing an OpenGL scene. If both threads are called to draw portions of a scene and they both try to call glClear before drawing, one thread's object may get erased by another thread's call to glClear.

The GLTHREAD sample assigns the glClear function to a single thread, and ensures that the other thread does not perform any drawing until glClear has been called. When a menu command message is sent to the main window, the application calls CreateThread twice to create two threads. Each thread calls GetDC(hwndMain) to obtain its own device context to the main window.

Then, each thread calls GLTHREAD's bSetupPixelFormat function to set up the pixel format and calls wglCreateContext to create a new OpenGL Rendering Context. Now, each thread has its own Rendering Context and both can call wglMakeCurrent to make its new OpenGL rendering context its (the calling thread's) current rendering context.

All subsequent OpenGL calls made by the thread are drawn on the device identified by the HDC returned from each thread's call to GetDC(). Now, because only one thread should call glClear, GLTHREAD has thread number one call it. The second thread is created "suspended" so it does nothing until a call to ResumeThread is made. After thread one has called glClear, it enables thread two to resume by calling ResumeThread with a handle to the second thread.

The procedure in the main thread that created the two other threads waits until both threads are finished before returning from the processing of the menu command message that is sent when the user selects the "Draw Waves" menu selection from the "Test Threads" menu. It will use the WaitForMultipleObjects function to do this.

Additional reference words: 3.10 3.50 4.00 95 GDI GRAPHICS THREADS
KBCategory: kbgraphic kbgraphic kbfile kbcode
KBSubcategory: codesam GdiDrwOpenGL

SAMPLE: Insert OLE Object Capabilities to the RichEdit Control

PSS ID Number: Q141549

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY

RichEdit control supports embedding and inplace-activating OLE Objects. To add these capabilities to a RichEdit control in your application, an independent service vendor (ISV) must implement certain OLE interfaces. The RichEdit sample shows how these interfaces are implemented and other implementation details that are required in conjunction with the OLE interfaces to embed and inplace-activate OLE objects.

MORE INFORMATION

An ISV must implement the following OLE interfaces to add the embedding and inplace-activating capabilities to the RichEdit control in their application:

- IRichEditOleCallback, which is used by a RichEdit control to retrieve OLE-related information from its client. This interface is passed by the client application to the RichEdit control via EM_SETOLEINTERFACE message. This is a RichEdit control's custom interface.

- IOleInPlaceFrame, which is used to control the container's top-level frame window. In addition, implementing this interface involves implementing IOleWindow and IOleInPlaceUIWindow interfaces also, because IOleInPlaceFrame is inherited from IOleInPlaceUIWindow, which in turn inherits IOleWindow.

There are a few caveats that are important to enable this feature work to properly:

- The Frame window, which will be the parent of the tool bar and status bar, needs to be created with the WS_CLIPCHILDREN style. If this style is not present, your applications will exhibit some painting problems when an object is inplace-active in your RichEdit control.

- The RichEdit control itself should be created with WS_CLIPSIBLING style. Here too, if the style is not present, RichEdit control will exhibit painting problems when the Object creates child windows during inplace-active.

- When destroying the RichEdit control, your application should deactivate

any inplace-active Object and call IOleObject->Close() on all the embedded objects in the RichEdit control. If this is not done, some object applications may not close down, thus causing them to stay in memory, even after the RichEdit control is destroyed. Here is a code snippet that demonstrates how to handle closing of OLE Objects:

```
   if (m_pRichEditOle)
   {
      HRESULT hr = 0;

      //
      // Start by getting the total number of objects in the control.
      //
      int objectCount = m_pRichEditOle->GetObjectCount();

      //
      // Loop through each object in the control and if active
      // deactivate, and if open, close.
      //
      for (int i = 0; i < objectCount; i++)
      {
         REOBJECT reObj;
          ZeroMemory(&reObj, sizeof(REOBJECT));
         reObj.cbStruct  = sizeof(REOBJECT);

         //
         // Get the Nth object
         //
         hr = m_pRichEditOle->GetObject(i,&reObj,REO_GETOBJ_POLEOBJ);
         if(SUCCEEDED(hr))
         {
            //
            // If active, deactivate.
            //
            if (reObj.dwFlags && REO_INPLACEACTIVE)
               m_pRichEditOle->InPlaceDeactivate();

            //
            // If the object is open, close it.
            //
            if(reObj.dwFlags&&REO_OPEN)
               hr = reObj.poleobj->Close(OLECLOSE_NOSAVE);

               reObj.poleobj->Release();
         }
      }
      m_pRichEditOle->Release();
   }
```

Download Richedit.exe, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- The Microsoft Network
  On the Edit menu, click Go To, and then click Other Location
  Type mssupport

Double-click the MS Software Library icon
Find the appropriate product area
Download Richedit.exe

- Microsoft Download Service (MSDL)
Dial (206) 936-6735 to connect to MSDL
Download Richedit.exe

- Internet (anonymous FTP)
ftp ftp.microsoft.com
Change to the Softlib\Mslfiles directory
Get Richedit.exe

Additional reference words: 4.00 1.30 softlib
KBCategory: kbui kbole kbhowto kbfile
KBSubcategory:

SAMPLE: KEYBFONT.EXE: Input Language and Font Matching

PSS ID Number: Q152753

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.5, 3.51, 4.0

SUMMARY

=======

In Windows 95 and Windows NT 4.0 non-Unicode applications, when the input language changes and, as a result, the default character set supported by this language also changes, the script of the current font should also be changed, such as changing the input language from English to Russian. By the same token, when the character set (script) of the font changes, the input language should be changed to match the new script, such as changing from Times New Roman Western script to Times New Roman Cyrillic script. Note that in Windows NT 4.0 Unicode applications, changing the font is not necessary because of support for multicharset fonts. For these applications, the methods described here can be used to detect if a given language is supported by a given font.

The KEYBFONT sample demonstrates the use of WM_INPUTLANGCHANGEREQUEST and WM_INPUTLANGCHANGE messages to match the current font to the input language. If the current font is changed, the list of loaded keyboard drivers is scanned for a match and, if one is found, the font is allowed to change. Otherwise, an error message is displayed and the font does not change. If the input language changes, the scripts of the current font are scanned for a match and, if none is found, an error message is displayed and the input language does not change.

Download KEYBFONT.EXE, a self-extracting file, from the following services:

- Microsoft's World Wide Web site on the Internet
  - On the www.microsoft.com home page, click the Support icon
  - Click Knowledge Base, and select the product
  - Enter kbfile KEYBFONT.EXE, and click GO!
  - Open the article, and click the button to download the file

- Internet (anonymous FTP)
  - ftp ftp.microsoft.com
  - Change to the Softlib/Mslfiles folder
  - Get KEYBFONT.EXE

- The Microsoft Network
  - On the Edit menu, click Go To, and then click Other Location
  - Type "mssupport" (without the quotation marks)
  - Double-click the MS Software Library icon
  - Find the appropriate product area
  - Locate and download KEYBFONT.EXE

- Microsoft Partner Network (MSPN)
  - On MSPN Desktop, double-click the Software Library icon

Search for KEYBFONT.EXE
Display results and download

- Microsoft Download Service (MSDL)
Dial (206) 936-6735 to connect to MSDL
Download KEYBFONT.EXE

For additional information about downloading, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q119591
TITLE : How to Obtain Microsoft Support Files from Online Services

NOTE: Use the -d option when running KEYBFONT.EXE to decompress the file and recreate the proper directory structure.

MORE INFORMATION
==========

The WM_INPUTLANGCHANGEREQUEST message is sent when the user chooses an input language, either with an input language change hotkey (see Control Panel.Keyboard.Language.Switch Languages) or from the language indicator on the system taskbar. This indicator will only be present if more than one keyboard layout has been installed and the indicator is enabled. An application can accept the change by passing the message to the DefWindowProc function, or reject the change, preventing it from taking place, by returning immediately.

In this sample, during WM_INPUTLANGCHANGEREQUEST, the character set of the requested language is compared to the default character set of each font script in the current font family via EnumFontFamilies(). If the font is not supported, the function returns immediately, disallowing any language change request.

The WM_INPUTLANGCHANGE message is sent to the top-most affected window after a task's locale has been changed. It should be used to make any application-specific settings, such as changing the current font script, and passed on to the DefWindowProc function to be passed on to any children.

In this sample, during WM_INPUTLANGCHANGE, the script of the current font is changed to match the character set of the language the user has switched to.

REFERENCES
==========

For more information on font scripts and multilanguage keyboard support, please see the Chapter "Accommodating Multilingual I/O on Microsoft Windows" in "Developing International Software", by Nadine Kano (MS Press ISBN: 1-55615-840-8).

Additional reference words: 4.00 softlib internationalization
KBCategory: kbprg kbprint kbui kbfile
KBSubcategory: WIntlDev

SAMPLE: MFCOGL a Generic MFC OpenGL Code Sample

PSS ID Number: Q127071

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0
- The Microsoft Foundation Classes (MFC) included with:
  - Microsoft Visual C++ 32-bit Edition, versions 2.0 and 2.1

SUMMARY

Microsoft Windows NT's OpenGL can be used with the Microsoft Foundation Class (MFC) library. This article gives you the steps to follow to enable MFC applications to use OpenGL.

The companion sample (MFCOGL) is a generic sample that demonstrates using OpenGL with MFC. Download MFCOGL.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download MFCOGL.EXE
- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get MFCOGL.EXE

MORE INFORMATION

Step-by-Step Example to Use OpenGL in MFC Application

1. Include the necessary header files to use OpenGL including:
   - "gl\gl.h" for all core OpenGL library fuctions. These functions have the "gl" prefix such as glBegin().
   - "gl\glu.h" for all OpenGL utility library functions. These functions have the "glu" prefix such as gluLookAt().
   - "gl\glaux.h" for all Windows NT OpenGL auxilary library functions. These functions have the "aux" prefix such as auxSphere().

   You don't need to add a header file for functions with the "wgl" prefix.

2. Add necessary library modules to the link project settings. These library modules include OPENGL32.LIB, GLU32.LIB, and GLAUX.LIB.

3. Add implementations for OnPaletteChanged() and OnQueryNewPalette() in CMainFrame class for palette-aware applications.

```
void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
{
    CFrameWnd::OnPaletteChanged(pFocusWnd);

    if (pFocusWnd != this)
        OnQueryNewPalette();

}

BOOL CMainFrame::OnQueryNewPalette()
{
    WORD   i;
    CPalette   *pOldPal;
    CMfcOglView *pView = (CMfcOglView *)GetActiveView();
    CClientDC   dc(pView);


    pOldPal = dc.SelectPalette(&pView->m_cPalette, FALSE);
    i = dc.RealizePalette();
    dc.SelectPalette(pOldPal, FALSE);

    if (i > 0)
        InvalidateRect(NULL);

    return CFrameWnd::OnQueryNewPalette();
}
```

4. Use the one or more of the following classes derived from CWnd, including view classes, that will use OpenGL for rendering onto:

   - Implement PreCreateWindow() and add WS_CLIPSIBLINGS and WS_CLIPCHILDREN to the windows styles:

```
BOOL CMfcOglView::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style |= WS_CLIPSIBLINGS | WS_CLIPCHILDREN;

    return CView::PreCreateWindow(cs);
}
```

   - Implement OnCreate() to initialize a rendering context and make it current. Also, initialize any OpenGL states here:

```
int CMfcOglView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;
```

```
    Init(); // initialize OpenGL

    return 0;
}
```

- Implement OnSize() if the window is sizeable:

```
void CMfcOglView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    if (cy > 0)
    {
        glViewport(0, 0, cx, cy);

        if ((m_oldRect.right > cx) || (m_oldRect.bottom > cy))
            RedrawWindow();

        m_oldRect.right = cx;
        m_oldRect.bottom = cy;

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45.0f, (GLdouble)cx / cy, 3.0f, 7.0f);
        glMatrixMode(GL_MODELVIEW);
    }
}
```

- Implement OpenGL rendering code. This can be done in OnDraw() or other application-specific places such as OnTimer().

- Implement clean-up code, which is typically done in OnDestroy():

```
void CMfcOglView::OnDestroy()
{
    HGLRC   hrc;

    if (m_nTimerID)
        KillTimer(m_nTimerID);

    hrc = ::wglGetCurrentContext();

    ::wglMakeCurrent(NULL,  NULL);

    if (hrc)
        ::wglDeleteContext(hrc);

    CPalette    palDefault;

    // Select our palette out of the dc
    palDefault.CreateStockObject(DEFAULT_PALETTE);
    m_pDC->SelectPalette(&palDefault, FALSE);

    if (m_pDC)
        delete m_pDC;
```

```
            CView::OnDestroy();
        }
```

Additional reference words: 3.50 3.51 2.00 2.10 3.00 3.10 4.00 95 graphics
KBCategory: kbgraphic kbcode kbfile
KBSubcategory: GdiOpenGL

SAMPLE: RASberry - an RAS API Demonstration

PSS ID Number: Q118983

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0

SUMMARY

=======

RASberry is a sample application that demonstrates the Remote Access Service (RAS) API. RASberry allows the user to enumerate current RAS connections, display the status of a selected connection, dial entries from the default phone book, and hang up an active connection. The sample may be built for both Windows version 3.1 and Win32 environments.

RASBRY.EXE can be downloaded as a self-extracting file from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download RASBRY.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get RASBRY.EXE

MORE INFORMATION

================

The RAS APIs used in this sample are:

- RasDial
- RasEnumConnections
- RasEnumEntries
- RasGetConnectStatus
- RasGetErrorString
- RasHangUp

The RAS SDK for Win32 is included as part of the Win32 SDK.

The RAS SDK for Windows version 3.1 (which contains additional files that can be used with the Windows version 3.1 SDK for RAS development) is available on the Microsoft Developer's Network Level 2 CD set, beginning with the April 1994 edition.

Additional reference words: 3.10 3.50 4.00 95 softlib

KBCategory: kbnetwork kbnetwork kbfile

KBSubcategory: NtwkRAS

SAMPLE: RESIZE App Shows How to Resize a Window in Jumps

PSS ID Number: Q123605

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5
  - Microsoft Windows 95 version 4.0

SUMMARY

Sometimes it's useful to have a window that can only be certain sizes. For example, Microsoft Word and Microsoft Visual C++ have toolbars that are resizable only to particular sizes that best fit the controls in the toolbar. When you do this, it's a good idea to give the user visual cues about the available window sizes. The RESIZE sample code shows by example how to modify the way Windows resizes a window so that when a user uses the mouse to resize the window the border jumps automatically to the next available size.

To obtain the RESIZE sample code, download RESIZE.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download RESIZE.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the SOFTLIB\MSLFILES directory
  Get RESIZE.EXE

MORE INFORMATION

When a user clicks the resizing border of a window, Windows enters a PeekMessage loop to capture all the mouse messages that occur until the left mouse button is released. While inside this loop, every time the mouse moves it moves the rectangle that shows the new window size to provide a visual cue to the user as to what the new window size will be.

The RESIZE sample code modifies the resizing operation by entering it's own message loop to capture the mouse messages until the left button is released. Instead of updating the rectangle every time a mouse move is received, the RESIZE code checks to see if the current mouse position would make the window size one of the possible window width and height sizes as defined by the application. By doing this, the RESIZE application provides more accurate visual cues about what the resizing operation will do.

The resizing operation is triggered by the WM_NCLBUTTONDOWN message both for Windows and the RESIZE application. When this message is received, a

message loop is entered to filter out all the mouse messages except for two, WM_MOUSEMOVE and WM_LBUTTONUP. When the WM_MOUSEMOVE message is received, the RESIZE application checks to see if the current mouse position would make the window larger or smaller. If the window would be smaller, the resizing rectangle is moved to the next smaller dimension defined by the application. If the window would be larger, the program checks to see if the new size would be large enough for the next possible dimension and updates the rectangle accordingly. When the WM_LBUTTONUP message is received, the resizing operation is completed by updating the window size to the current position defined by the mouse and the rectangle is removed.

The RESIZE application also takes advantage of some of the flexibility provided by processing the WM_NCHITTEST message. Windows sends this message to an application with a mouse position and expects the application to describe which part of the window that mouse position covers. Frequently, applications pass this message on to DefWindowProc() and let the default calculations take care of telling the system what the mouse is on top of. The RESIZE application allows DefWindowProc() to process the message, but then checks to see if the mouse is over one of the resizing corners or in the client area. To simplify the resizing operation, RESIZE doesn't let the user resize from a window corner, so the application overrides the HTBOTTOMLEFT, HTBOTTOMRIGHT, HTTOPLEFT, and HTTOPRIGHT hit test codes and returns HTBOTTOM or HTTOP. By doing this, the mouse cursor accurately reflects the direction of the resize. When the HTCLIENT hit test code is returned, RESIZE changes this to HTCAPTION to allow the window to be moved even though it doesn't have a title bar.

Although this technique will work in Windows 95, it is not necessary. Windows 95 provides a new message WM_SIZING that will enable the program to do exactly the same thing without processing the WM_NCxxx messages or entering a PeekMessage() loop.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbui kbcode kbfile
KBSubcategory: UsrWndw

SAMPLE: SCLBLDLG - Demonstrates Scaleable Controls in Dialog

PSS ID Number: Q112639

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

In certain circumstances it is desirable to dynamically scale the controls in a dialog box to the size of the dialog box window. SCLBLDLG.EXE is a file in the Microsoft Software Library that contains sample code implementing scaleable controls in a dialog box.

SCLBLDLG can be downloaded as a self-extracting file from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download SCLBLDLG.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get SCLBLDLG.EXE

MORE INFORMATION

================

To accomplish scaleable controls in a dialog box, the following messages are processed:

WM_INITDIALOG - To store original dimensions of the dialog box and all its controls, together with the font the dialog box uses. The original dimensions are stored using SetProp() in this sample. The font handle is stored in a static variable to be used with in WM_SIZE.

WM_SIZE - To calculate the scaling factor, and then scale up or down the font and all the controls in the dialog box.

WM_GETMINMAXINFO - To set the minimum size of the dialog box so the controls are not truncated.

WM_COMMAND - To clean up when closing the dialog box. RemoveProp() is called to remove the stored dimensions from the property list of the dialog box window and all its child control windows.

NOTE: Special processing is required for calculating the dimensions of CBS_DROPDOWN and CBS_DROPDOWNLIST style combo boxes. GetWindowRect() returns the dimensions of the edit portion of the combo box, excluding the drop-down list. To get the correct height for such combo boxes, the value returned by CB_GETDROPPEDCONTROLRECT is used instead of GetWindowRect().

Additional reference words: 3.10 3.50 3.51 4.00 95 proportional sizing softlib
KBCategory: kbui kbfile
KBSubcategory: UsrDlgs

SAMPLE: Setting Tab Stops in a Windows List Box

PSS ID Number: Q66652

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

Tab stops can be used in a list box to align columns of information. This article describes how to set tab stops in a list box and provides a code example that demonstrates the process.

MORE INFORMATION
================

To set tab stops in a list box, perform the following three steps:

1. Specify the LBS_USETABSTOPS style when creating the list box.

2. Assign the desired tab stops to an integer array.

   a. The tab stop values must be in increasing order -- back tab stops are not allowed. The tabs work the same as typewriter tabs: once a tab stop is overrun, a tab character will move the cursor to the next tab stop. If the tab stop list is overrun (that is, the current position is greater than the last tab stop value), the default tab of eight characters is used.

   b. The tab stops should be specified in dialog units. On the average, each character is about four horizontal dialog units in width.

   c. It is possible to hide columns of text from the user by specifying tab stops beyond the right side of the list box. This can be a useful way to hide information used for the application's internal processing.

3. Send an LB_SETTABSTOPS message to the list box to set the tab stops. For example, in Windows 3.1:

```
SendMessage(GetDlgItem(hDlg, IDD_LISTBOX),
            LB_SETTABSTOPS,
            TOTAL_TABS,
            (LONG)(LPSTR)TabStopList);
```

   a. If wParam is set to 0 (zero) and lParam to NULL, the tab stops are

set to two dialog units by default.

b. SendMessage() will return TRUE if all of the tab stops are set successfully; otherwise, SendMessage() returns FALSE.

Example

Below is an example of the process. Tab stops are set at character positions 16, 32, 58, and 84.

```
int     TabStopList[TOTAL_TABS]; /*  Array to store tabs */

TabStopList[0] = 16 * 4;          /* 16 spaces */
TabStopList[1] = 32 * 4;          /* 32 spaces */
TabStopList[2] = 58 * 4;          /* 58 spaces */
TabStopList[3] = 84 * 4;          /* 84 spaces */

SendMessage(GetDlgItem(hDlg, IDD_LISTBOX),
            LB_SETTABSTOPS,
            TOTAL_TABS,
            (LONG)(LPSTR)TabStopList);
```

NOTE: For Win32, use LPARAM instead of LONG.

If the desired unit of measure is character position, then specifying tab positions in dialog units is recommended. Dialog units are independent of the current font; they are loosely based on the average width of the system font. Each character takes approximately four dialog units.

NOTE: Under Windows 95, dialog base units for dialogs based on non-system fonts are calculated in a different way than under Windows 3.1. For more information, please see the following article in the Microsoft Knowledge Base:

```
ARTICLE-ID: Q125681
TITLE     : How to Calculate Dialog Base Units with Non-system-based
            Font
```

For more control over the exact placement of a tab stop, the desired position should be converted to a pixel offset and this offset should be converted into dialog units. The following formula will take a pixel position and convert it into the first tab stop position before (or at) the desired pixel position:

```
TabStopList[n] = 4 * DesiredPixelPosition /
                 LOWORD(GetDialogBaseUnits());
```

There is a sample application named TABSTOPS in the Microsoft Software Library that demonstrates how tab stops are set and used in a list box.

Download TABSTOPS.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
    Dial (206) 936-6735 to connect to MSDL
    Download TABSTOPS.EXE

- Internet (anonymous FTP)
    ftp ftp.microsoft.com
    Change to the \SOFTLIB\MSLFILES directory
    Get TABSTOPS.EXE

Additional reference words: 3.00 3.10 3.50 4.00 softlib
KBCategory: kbui kbfile
KBSubcategory: UsrCtl

SAMPLE: Simulating Palette Animation on Non-Palette Displays

PSS ID Number: Q130476

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SUMMARY

LAVALAMP is a sample application in the Microsoft Software Library that demonstrates how to simulate the effects of the AnimatePalette() function on devices that may not support palettes. This program also demonstrates how to create and manipulate dibsections. The following dibsection functions are used in LAVALAMP:

```
CreateDIBSection()
GetDIBColorTable()
SetDIBColorTable()
```

Download LAVALAMP.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download LAVALAMP.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get LAVALAMP.EXE

MORE INFORMATION

When running in display modes that are not palette-based, many of the effects that can be performed easily with palette animation need to be reprogrammed. A simple method of simulating palette animation can be achieved by "animating" a device-independent bitmap's (DIBs) color table and redisplaying the DIB with the new colors. To demonstrate this technique, LAVALAMP creates an 8-bits-per-pixel (bpp) dibsection. Then it shifts each of the RGBQUAD data structures in the color table by one position to the left, and recycles the first entry in the color table to the last position. After each modification to the color table, the DIB is redisplayed.

Because the entire DIB must be redisplayed after each modification to the color table, this technique is not recommended for large bitmaps.

Additional reference words: 3.50 technote BMP softlib
KBCategory: kbgraphic kbfile

KBSubcategory: GdiBmp GdiPal

SAMPLE: The Palette Manager: How and Why It Does What It Does

PSS ID Number: Q137372

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1, 3.11
- Microsoft Win32 SDK versions 3.5, 3.51, 4.0

You can now download a document that gives a full introduction to the Microsoft Windows Palette Manager along with two samples that accompany it. Beyond simply describing how to use the palette interface, this document explains its specific internal workings and gives reasons for its chosen implementation. Due to the complexity of the Palette Manager, this document introduces some topics and then goes into them in greater detail. It is best to read it from start to finish before using it as a reference.

After downloading the file, use -d when running it so that the samples are placed their own directories. In other words, place Palman.exe in an empty directory and run it using this command:

```
Palman.exe -d
```

Download Palman.exe, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download Palman.exe

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get Palman.exe

Additional reference words: 3.50 4.00
KBCategory: kbref kbgraphic kbfile
KBSubcategory: W32

SAMPLE: Using Blinking Text in an Application

PSS ID Number: Q11787

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

It is possible to create blinking text in a Windows-based application. Because there are no character attributes similar to the normal MS-DOS text environment, the application must repeatedly paint the screen to implement the flash. This article, and an accompanying file in the Microsoft Software Library, demonstrate how this is done.

MORE INFORMATION

A timer is used to determine the rate at which the text flashes. Timer messages are processed by inverting the appropriate area in the window using the DSTINVERT action of the PatBlt function. The second time that the PatBlt function is called, the text returns to its original state. Alternatively, the PATINVERT action of the PatBlt function may be used. To use this method, an appropriate brush must be selected into the display context as the current pattern. This method requires more work, however, it is more flexible.

The rate at which the text blinks can be set to match the cursor blink time set in the Control Panel. To do this, the following code should be run when the application starts and in response to WM_WININICHANGE messages:

```
   nRate = GetProfileInt(
     (LPSTR)"windows",         /* heading in [] */
     (LPSTR)"CursorBlinkRate", /* string to match */
      550);                    /* default value */
```

Be sure to delete the timer when the application terminates.

There is a sample program in the Microsoft Software Library named BLINK that uses this technique to demonstrate blinking text.

Download BLINK.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL

Download BLINK.EXE

- Internet (anonymous FTP)
  Download BLINK.EXE
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get BLINK.EXE

Additional reference words: 3.00 3.10 3.50 4.00 95 softlib BLINK.EXE
KBCategory: kbui kbfile
KBSubcategory: UsrPnt

SAMPLE: Win16 App (WOW & Win32s) Calling Win32 DLL Code

PSS ID Number: Q114341

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SUMMARY

=======

The INTEROP sample demonstrates two general methods for calling routines in a Win32 DLL from a Windows-based application: thunks and SendMessage(). There are two different thunking methods, one for each platform: Generic Thunks on Windows NT and Universal Thunks on Win32s. The message used is WM_COPYDATA, a new message introduced by Windows NT and Win32s. All three methods provide a way to call functions and pass data across the 16-32 boundary, translating any pointers in the process. The advantages of WM_COPYDATA over thunks are that it is fast the exact same code runs on either platform (the thunk used will depend on the platform). The disadvantage of WM_COPYDATA is that a method must be devised to get a function return value (other than true or false) back to the calling application.

NOTE: Universal Thunks were designed to work with a Win32-based application calling a 16-bit DLL. The method described here has limitations. Because the application is 16-bit, no 32-bit context is created, so certain calls will not work from the Win32 DLL.

The sample consists of the following source files:

APP16.C	-	Win16 Application
DLL16.C	-	16-bit side of Universal Thunk/Generic Thunk
STUB32.C	-	32-bit stub that loads the 32-bit DLLs on Win32s
UTDLL32.C	-	32-bit side of the Universal Thunk
DISP32.C	-	Dispatch calls sent through WM_COPYDATA
DLL32.C	-	Win32 DLL

This sample is included with the Microsoft Win32 SDK. It is located in SCT\SAMPLES\INTEROP. NOTE: There is also an RPC sample named INTEROP, but it is in a different directory.

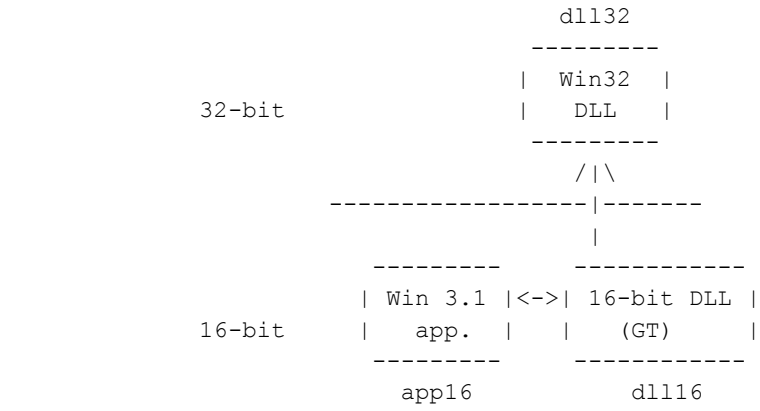
MORE INFORMATION

================

Generic Thunk

Under Windows NT, it is possible to call routines in a Win32 DLL from a Windows-based application using an interface called Generic Thunks. The SDK file DOC\SDK\MISC\GENTHUNK.TXT describes the interface.

Here is a picture of the way the pieces fit together in INTEROP:



DLL16 is loaded when APP16 is loaded. If it detects that WOW is present, then it loads DLL32.

WOW presents a few new 16-bit application programming interfaces (APIs) that allow you to load the Win32 DLL, get the address of the DLL routine, call the routine (passing it up to thirty-two 32-bit arguments), convert 16:16 (WOW) addresses to 0:32 addresses (useful if you need to build up a 32-bit structure that contains pointers and pass a pointer to it), and free the Win32 DLL. These functions are:

```
DWORD FAR PASCAL LoadLibraryEx32W( LPCSTR, DWORD, DWORD );
DWORD FAR PASCAL GetProcAddress32W( DWORD, LPCSTR );
DWORD FAR PASCAL CallProc32W( DWORD, ..., LPVOID, DWORD, DWORD );
DWORD FAR PASCAL GetVDMPointer32W( LPVOID, UINT );
BOOL FAR PASCAL FreeLibrary32W( DWORD );
```

When linking the Win16 application, you need to put the following statements in the .DEF file, indicating that the functions will be imported from the WOW kernel:

```
IMPORTS
   kernel.LoadLibraryEx32W
   kernel.FreeLibrary32W
   kernel.GetProcAddress32W
   kernel.GetVDMPointer32W
   kernel.CallProc32W
```

Note that although these functions are called in 16-bit code, they need to be provided with 32-bit handles, and they return 32-bit handles.

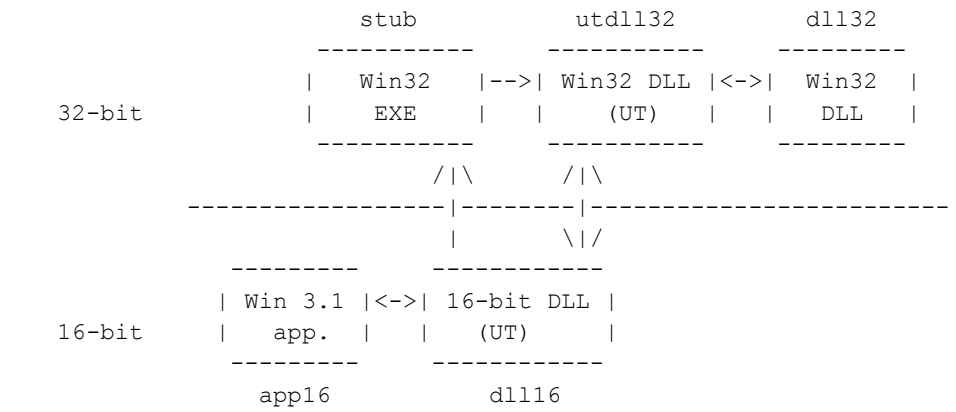
In addition, be sure that your Win32 DLL entry points are declared with the _stdcall convention; otherwise, you will get an access violation.

Universal Thunk

Under Win32s, it is possible to call routines in a Win32 DLL from a Win16 application using an interface called Universal Thunks. The interface is described in the Win32s Programmer's Reference. The sample UTSAMPLE, shows

the opposite (and more typical) case, a Win32 application calling 16-bit routines.

Here is a picture of the way the pieces fit together in INTEROP:



The load order is as follows: The Windows 3.1 application loads the 16-bit DLL. The 16-bit DLL checks to see whether the 32-bit side has been initialized. If it has not been initialized, then the DLL spawns the 32-bit .EXE (stub), which then loads the 32-bit DLL that sets up the Universal Thunks with the 16-bit DLL. Once all of the components are loaded and initialized, when the Windows 3.x application calls an entry point in the 16-bit DLL, the 16-bit DLL uses the 32-bit Universal Thunk callback to pass the data over to the 32-bit side. Once the call has been received on the 32-bit side, the proper Win32 DLL entry point can be called.

WM_COPYDATA

The wParam and lParam for this message are as follows:

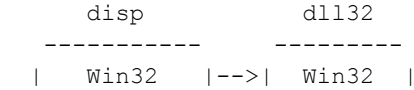
```
wParam = (WPARAM) (HWND) hwndFrom;   /* handle of sending window */
lParam = (LPARAM) (PCOPYDATASTRUCT) pcds;
```

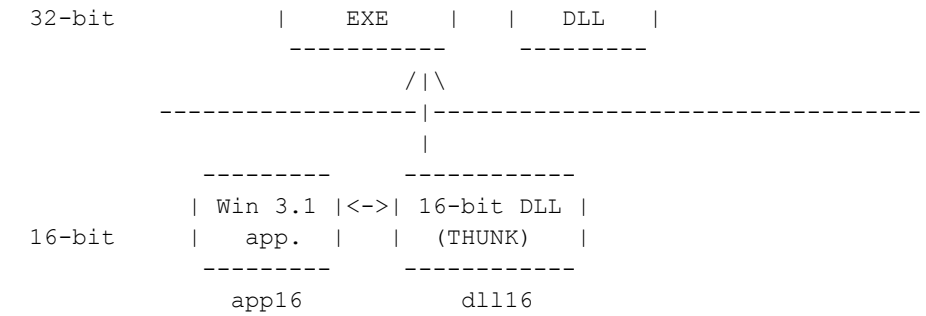
Where hwndFrom is the handle of the sending window and COPYDATASTRUCT is defined as follows:

```
typedef struct tagCOPYDATASTRUCT {
    DWORD dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT;
```

The INTEROP sample uses dwData as a function code, indicating which Win32 DLL entry point should be calling and lpData to contain a pointer to the data structure to be passed to the function.

Here is a picture of the way the pieces fit together in INTEROP:





DLL16 is loaded when APP16 is loaded. DISP is spawned to handle WM_COPYDATA messages, regardless of platform. DISP dispatches the calls to DLL32, marshalling the arguments.

Additional reference words: 3.50
KBCategory: kbref
KBSubcategory: SubSys

Secure Erasure Under Windows NT

PSS ID Number: Q94239

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY
=======

File systems under Windows NT currently have virtual secure erasure (when a file is deleted, the data is no longer accessible through the operating system). Although the bits could still be on disk, Windows NT will not allow access to them.

MORE INFORMATION
================

The NTFS file system does this by keeping a high-water mark, for each file, of bytes written to the file. Everything below the line is real data, anything above the line is (on disk) random garbage that used to be free space, but any attempt to read past this high-water mark returns all zeros.

Other reusable objects are also protected. For example, all the memory pages in a process's address space are zeroed when they are touched (unlike the file system, a process may directly access its pages, and thus the pages must be actually zeroed rather than virtually zeroed).

Note that file system security assumes physical security; in other words, if a person has physical access to a machine and can boot an alternative operating system and/or add custom device drivers and programs, he/she can always get direct access to the bits on disk.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Security and Screen Savers

PSS ID Number: Q96780

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY
=======

Screen savers are user-mode applications that execute in a different desktop (see the note at the end of the article). Therefore, a screen saver cannot enumerate windows of user-mode applications. This design prevents unauthorized users from viewing the contents of applications displayed on the screen. For secure screen savers (those that ask for a password), this adds a further layer of protection.

Screen savers also execute in the security context of the logged-on user. A screen saver may call ExitWindowsEx(), to log off from or shut down the system, or any other application programming interface (API) that the logged-on user has permission to perform.

MORE INFORMATION
================

A sample screen saver SCRNSAVE is distributed on the Win32 SDK CD.

NOTE: A desktop is a virtual screen. Windows NT 3.5 and earlier have three desktops--the main desktop, the WinLogon desktop, and a desktop for screen savers. Windows NT 3.51 and later support and document creating multiple desktops.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Security Attributes on Objects

PSS ID Number: Q102798

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

Early betas of Windows NT did not require security attributes on objects such as pipes. For example, it was valid at that time to enter NULL for the last parameter of the Win32-based application programming interface (API) CreateNamedPipe(). This is no longer the case.

MORE INFORMATION

Windows NT 3.1 and later require security attributes. Please note that setting the security attributes parameter to NULL does not indicate that you want a NULL security descriptor (SD), rather it indicates that you want to inherit the security descriptor of the current access token. For example, this means that any client wanting to connect to your pipe server must have the same security attributes as the user that started the server. If the user who started the server was the administrator of the machine, then any client who wants to connect must also be an administrator for that machine.

Below is an code sample that demonstrates creating a named pipe with a NULL security descriptor.

```
HANDLE               hPipe;    // Pipe handle.
SECURITY_ATTRIBUTES  sa;       // Security attributes.
PSECURITY_DESCRIPTOR pSD;      // Pointer to SD.

// Allocate memory for the security descriptor.

pSD = (PSECURITY_DESCRIPTOR) LocalAlloc(LPTR,
                             SECURITY_DESCRIPTOR_MIN_LENGTH);

// Initialize the new security descriptor.

InitializeSecurityDescriptor(pSD, SECURITY_DESCRIPTOR_REVISION);

// Add a NULL descriptor ACL to the security descriptor.

SetSecurityDescriptorDacl(pSD, TRUE, (PACL) NULL, FALSE);

sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = pSD;
sa.bInheritHandle = TRUE;
```

```
   // Create a local named pipe with a NULL security descriptor.

   hPipe = CreateNamedPipe(
         "\\\\.\\PIPE\\test",    // Pipe name = 'test'.
         PIPE_ACCESS_DUPLEX      // 2-way pipe.
         | FILE_FLAG_OVERLAPPED, // Use overlapped structure.
         PIPE_WAIT               // Wait on messages.
         | PIPE_READMODE_MESSAGE // Specify message mode pipe.
         | PIPE_TYPE_MESSAGE,
         MAX_PIPE_INSTANCES,     // Maximum instance limit.
         OUT_BUF_SIZE,           // Buffer sizes.
         IN_BUF_SIZE,
         TIME_OUT,               // Specify time out.
         &sa);                   // Security attributes.
```

It is important to note that by specifying TRUE for the fDaclPresent parameter and NULL for pAcl parameter of the SetSecurityDescriptorDacl() API, a NULL access control list (ACL) is being explicitly specified.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Security Context of Child Processes

PSS ID Number: Q111545

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

When a process is created with the CreateProcess() API, it operates under the same security context as the parent. The child process inherits the parent's security context, which are defined by the parent's access token. Even if a thread in the parent process impersonates a client and then creates a new process, the new process still runs under the parent's original security context and not the under the impersonation token.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Semicolons Cannot Separate Macros in .HPJ File

PSS ID Number: Q83020

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

The author of a Help system can combine individual macro commands into a macro string, which the Microsoft Help Compiler processes as a unit. When the macro string is part of an RTF text file, the individual macros in the string are separated from each other with a semicolon character (;). The Help system runs the individual macros of a macro string sequentially.

The author can define a macro string that is run when the user loads a Help file. This macro string is placed into the [CONFIG] section of the Help project (.HPJ) file. However, in the .HPJ file, the individual macros of the string are separated from each other with a colon character (:) because the semicolon character indicates the beginning of a comment.

MORE INFORMATION

In the following sample [CONFIG] section, a macro adds two buttons to the Help window's button bar. The first button is labeled "Other", which when chosen brings up the About Help dialog box. The second button is labeled "Test". When chosen, it disables the "Other" button and jumps to the topic represented by "context_string." To create the "Test" button, two macros are concatenated to form the macro parameter in the CreateButton call.

```
[CONFIG]
; This first button is added so that the demonstration macro is
; complete. This macro just creates a button. Choosing the button
; brings up the About Help dialog box.

CreateButton("other_button","&Other","About()")

; This macro also creates a button. Choosing the button disables
; "other_button", created above, and jumps to the topic represented by
; "topic_string."
;
; Note that the two macros in the CreateButton macro are separated by
; a colon, not a semicolon.
;
; NOTE: The following macro should appear on a single line.
```

```
CreateButton("test_button","&Test","DisableButton(`Other_Button'):
JumpId(`testhelp.hlp',`context_string')")
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsHlp

SendMessage() in a Multithreaded Environment

PSS ID Number: Q95000

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

When thread X calls SendMessage() to send a message to a window created by thread Y, it must wait until thread Y calls PeekMessage(), GetMessage(), or WaitMessage() before the SendMessage() call can continue. This process prevents synchronization problems.

MORE INFORMATION

Because of the multithreaded environment of Windows NT, SendMessage() does not behave in the same manner as it does under Windows 3.1. Under Windows 3.1, SendMessage() simply calls the window procedure given to it. In Windows 3.1, if you use the SendMessage() function to send a message to a window of another task, the system will perform a task switch to the target window (change to the stack of the target Windows-based application), let the target window process the message [when it calls GetMessage() or PeekMessage()], and then switch back to the original message.

Under Windows NT, however, only the thread that created a window may process the window's messages. Therefore, if thread X sends a message [via SendMessage()] to a window that was created by thread Y, thread X must wait for thread Y to be in a receiving state, and handle the message for it.

Thread Y is only in a receiving state when it calls PeekMessage(), GetMessage(), or WaitMessage(), because synchronization problems may occur if a thread is interrupted while processing other messages. While in a receiving state, thread Y may process messages sent to its windows via SendMessage() (in this case, by thread X).

Note that PeekMessage() and GetMessage() look in thread Y's message queue for messages. Because SendMessage() does not post any messages, PeekMessage() and GetMessage() will not see any indication of the SendMessage() call. The two functions merely serve as a point in time at which SendMessage() (thread X) may "interrupt," and have thread Y process its message next. Then PeekMessage() or GetMessage() continues normal operation under thread Y.

Because of this behavior, if thread X sends a message to thread Y, and thread Y is locked in a tight loop, thread X is now locked as well. This may be prevented by using SendNotifyMessage(), which behaves as SendMessage() does above, but returns immediately. This may be an advantage

if it is not important that the sent message be completed before thread Y continues. Note, however, that because SendNotifyMessage() is asynchronous, thread X should not pass pointers to any of its local variables when making the call, because they may be gone by the time thread Y attempts to look at them. This would result in a general protection violation (GP fault) when thread Y accesses the pointer.

Additional reference words: 3.10 3.50 3.51 4.00 95 GP-fault
KBCategory: kbui
KBSubcategory: UsrMisc

Service Control Manager Records Event ID 7023 In the System Log

PSS ID Number: Q133010

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51

SUMMARY

=======

The Service Control Manager records an event id 7023 into the system log when a service calls the SetServiceStatus API with the SERVICE_STATUS structure initialized in the following manner: the dwCurrentState member is equal to SERVICE_STOPPED and dwWin32ExitCode member is not equal to zero.

MORE INFORMATION

================

In the entry written into the system log, the event id is 7023 and the type of the event is ERROR. The source of the entry is the Service Control Manager. The description says, "<name of your service> terminated with the following error: <description of the error>"

Additional reference words:

KBCategory: kbprg

KBSubcategory: BseService

Services and Redirected Drives

PSS ID Number: Q115848

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

When writing a service, do not use WNetAddConnection() to redirect drives to remote shares, because the redirector does not allow the logged-on user to see the server or enumerate the drive. Instead, use universal naming convention (UNC) names whenever possible to redirect drives to remote shares.

MORE INFORMATION
================

WNetAddConnection() creates a global symbolic link describing the drive. Therefore, the user can change to the drive at the command prompt. However, WNetOpenEnum(), WNetEnumResource, and the "net use" command fail to list the drive connection that was created by the service. This happens because the logged-on user does not have an active session to the remote connection, so the redirector will not let the user see the remote server.

The File Manager calls GetDriveType() on each drive and puts up an icon for each drive found. The File Manager creates an icon for redirected drives created from a service because there is a global symbolic link to that drive. However, the share name is not available. In addition, the logged-on user cannot use File Manager to disconnect the drive because the drive was created by the service, not the logged-on user.

If the service process is running in the Local System account, then only that process or another process running in the Local System account can call WNetCancelConnection() to disconnect the drive.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseService

SetActiveWindow() and SetForegroundWindow() Clarification

PSS ID Number: Q97925

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

By default, each thread has an independent input state (its own active window, its own focus window, and so forth). SetActiveWindow() always logically sets a thread's active window state. To force a window to the foreground, however, use SetForegroundWindow(). SetForegroundWindow() activates a window and forces the window into the foreground. SetActiveWindow() always activates, but it brings the active window into the foreground only if the thread is the foreground thread.

NOTE: If the target window was not created by the calling thread, the active window status of the calling thread is set to NULL, and the active window status of the thread that created the target window is set to the target window.

Applications can call AttachThreadInput() to allow a set of threads to share the same input state. By sharing input state, the threads share their concept of the active window. By doing this, one thread can always activate another thread's window. This function is also useful for sharing focus state, mouse capture state, keyboard state, and window Z-order state among windows created by different threads whose input state is shared.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

SetBkColor() Does Not Support Dithered Colors

PSS ID Number: Q69885

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The syntax for the SetBkColor function is documented in the Microsoft Windows Software Development Kit (SDK) as follows:

```
DWORD SetBkColor(HDC hDC, COLORREF crColor);
```

SetBkColor sets the current background color of the specified device context (DC) to the color that the crColor parameter references, or to the nearest physical color if the device cannot represent the RGB color value that the crColor parameter specifies. In other words, SetBkColor cannot be used to set the background to a dithered color and defaults to the physical color that is closest to the requested crColor value.

MORE INFORMATION

This behavior can cause unexpected results for an application that changes the background color of a control to a color that cannot be represented by a color provided by the display device.

Specifically, when an application specifies a dithered color for the background of an edit control, and specifies the same color for the text background, Windows paints the control in two distinct colors.

For example, using the standard VGA display driver, the following call, in which COLOR_INACTIVEBORDER is a green/gray specified by RGB(64, 128, 128), sets the background color to gray (RGB(128, 128, 128)) rather than the dithered green/gray that is desired:

```
SetBkColor(wParam, GetSysColor(COLOR_INACTIVEBORDER));
```

To illustrate, if the application uses the function call while processing the WM_CTLCOLOR message to change the color of an edit control, the window background is painted green/gray, and the text background defaults to the nearest physical color, which is gray. This produces a gray rectangle inside a green/gray rectangle rather than the desired green/gray for the entire edit control.

This behavior can also occur with other controls such as option buttons and list boxes. However, an application can avoid this problem by using the SetBkMode function to set the background mode to TRANSPARENT. This allows the dithered brush pattern to show through beneath the text to achieve the desired results. That solution is not practical with a multiline edit control because if text is inserted, and the background mode has been set to TRANSPARENT, the text that is pushed to the right by the inserted text leaves its image behind. The result is text superimposed on top of other text, which quickly becomes unreadable.

To partially work around this situation for a multiline edit control, use the GetNearestColor function to determine the nearest physical color to the desired color, as in the code fragment below. In this case, the entire edit control is gray:

```
   case WM_CREATE:
      {
      HDC hDC;
      hDC = GetDC(hWnd);
      hGrayBrush = CreateSolidBrush(GetNearestColor(hDC,
            RGB(64, 128, 128)));
      ReleaseDC(hWnd, hDC);
      hWndEdit = CreateWindow( ... ES_MULTILINE ... );
      }
      break;

   case WM_CTLCOLOR:
      if (HIWORD(lParam) == CTLCOLOR_EDIT)
         {
         // The following call creates the nearest physical
         // color; therefore, it will be the same as the
         // hGrayBrush created above.
         SetBkColor(wParam, RGB(64, 128, 128));
         SetTextColor(wParam, RGB(255, 0, 0)); // red text
         return (DWORD)hGrayBrush;
         }
      else
        return DefWindowProc(hWnd, identifier, wParam, lParam);
      break;
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiDrw

SetClipboardData() and CF_PRIVATEFIRST

PSS ID Number: Q24252

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The documentation for SetClipboardData() states that CF_PRIVATEFIRST can be used to put private data formats on the clipboard. It also states that data of this format is not automatically deleted. However, that is apparently not true. That is, the data is removed automatically when the clipboard is emptied (sending a WM_DESTROYCLIPBOARD message) and the new item is set into the clipboard. The old item is shown; however, the handle now is invalid because GlobalFree() is called on it.

MORE INFORMATION

GlobalFree() was not called on this handle. If you try to use the other handle to this memory, you will find that the one you initially received from GlobalAlloc() is still valid. Only the clipboard handle has been invalidated by the call to EmptyClipboard().

The documentation states that "Data handles associated (with CF_PRIVATEFIRST) will not be freed automatically." This statement refers to the memory associated with that data handle. When SetClipboardData() is called under standard data types, it frees the block of memory identified by hMem. This is not the case for CF_PRIVATEFIRST. Applications that post CF_PRIVATEFIRST items on the clipboard are responsible for the memory block containing those items.

This is not intended to imply that items placed on the clipboard will remain on the clipboard if they are CF_PRIVATEFIRST. When a call is made to EmptyClipboard(), all objects will be removed.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrClp

SetErrorMode() Is Inherited

PSS ID Number: Q105304

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

An application can use SetErrorMode() to control whether the operating system handles serious errors or whether the application itself will handle the errors.

NOTE: The error mode will be inherited by any child process. However, the child process may not be prepared to handle the error return codes. As a result, the application may die during a critical error without the usual error message popups occurring.

This behavior is by design.

One solution is to call SetErrorMode() before and after the call to CreateProcess() in order to control the error mode that is passed to the child. Be aware that this process must be synchronized in a multithreaded application.

There is another solution available in Windows NT 3.5 and later. CreateProcess() has a new flag CREATE_DEFAULT_ERROR_MODE that can be used to control the error mode of the child process.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseExcept

SetParent and Control Notifications

PSS ID Number: Q104069

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

An edit, list box, or combo box control sends notifications to the original parent window even after SetParent has been used to change the control's parent. A button control sends notifications to the new parent after SetParent has been used to change its parent.

Edit, list box, and combo box controls keep a private copy of the window handle of the parent at the time of creation. This handle is not changed when SetParent is used to change the control's parent. Consequently, the notifications (EN_*, LBN_*, and CBN_* notifications) go to the original parent.

Note that WM_PARENTNOTIFY messages go to the new parent and GetParent() returns the new parent. If it is required that notifications go to the new parent window, code must be added to the old parent's window procedure to pass on the notifications to the new parent.

For example:

```
case WM_COMMAND:
  hwndCtl = LOWORD(lParam);

  // If notification is from a control and the control is no longer this
  //   window's child, pass it on to the new parent.
  if (hwndCtl && !IsChild(hWnd, hwndCtl))
      SendMessage(GetParent(hwndCtl), WM_COMMAND, wParam,  lParam);
  else Do normal processing;
```

Button controls send notifications to the new parent after SetParent has been used to change the parent.

Additional reference words: 3.10 3.50 3.51 4.00 95 listbox combobox
KBCategory: kbui
KBSubcategory: UsrCtl

SetTimer() Should Not Be Used in Console Applications

PSS ID Number: Q102482

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

SetTimer() was not designed to be used with a console application because it requires a message loop to dispatch the timer signal to the timer procedure. In a console application, this behavior can be easily emulated with a thread that is set to wait on an event.

MORE INFORMATION

In Windows NT 3.1, SetTimer() can work within a console application, but it requires a thread in a loop calling GetMessage() and DispatchMessage().

For example,

```
   while (1)
   {
      GetMessage();
      DispatchMessage();
   }
```

Because this requires a thread looping, there is no real advantage to adding a timer to a console application over using a thread waiting on an event.

Another option is to use a multimedia timer, which does not require a message loop and has a higher resolution. In Windows NT 3.5, the resolution can be set to 1 msec using timeBeginPeriod(). See the help for timeSetEvent() and the Multimedia overview. Any application using Multimedia calls must include MMSYSTEM.H, and must link with WINMM.LIB.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMisc

Setting Dynamic Breakpoints in WinDbg

PSS ID Number: Q100642

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SUMMARY
=======

The WinDbg breakpoint command contains a metacircular interpreter; that is, you can execute commands dynamically once a breakpoint is hit. This allows you to perform complex operations, including breaking when an automatic variable has changed, as described below.

The command interpreter of WinDbg allows any valid C expression to serve as a break condition. For example, to break whenever a static variable has changed, use the following expression in the Expression field of the breakpoint dialog box:

```
&<variablename>
```

In addition, the length should be specified as 4 (the size of a DWORD) in the length field.

This technique does not work for automatic variables because the address of an automatic variable may change depending on the value that the stack pointer has upon entering the function that defines the automatic variable. This is one case where the breakpoint needs to be redefined dynamically.

For this purpose, a breakpoint can be enabled at function start and disabled at function exit, so that the address of the variable is recomputed.

MORE INFORMATION
================

Suppose that the name of the function is "subroutine" and the local variable name is "i". The following steps will be used:

1. Start the program and step into the function that defines the automatic variable with the commands:

   ```
   g subroutine
   p
   bp500 ={subroutine}&i /r4 /C"?i"
   ```

   The breakpoint number is chosen to be large so that the breakpoint will be well out of range of other breakpoints. Note that /r4 indicates a length of 4 because i is an integer. Make this number larger for other data types. The command "?i" prints out the value of i.

2. Next, disable this first breakpoint with the command

   bd500

   because the address of i may change. The breakpoint will be enabled
   when in the scope of function subroutine.

3. The second breakpoint definition is set at the entry point of the
   function:

   bp .<FirstLine> /C"be 500;g"

   This is where the breakpoint is enabled. Note that <FirstLine> is
   the line number of the first statement in the function subroutine.

4. The last breakpoint is set at the end of the function

   bp .<LastLine> /C"bd 500;g"

   and will disable the breakpoint again. Note that <LastLine> is the
   line number of the last statement in the function subroutine.

   Note that if the function has more than one exit point, multiple
   breakpoints may have to be defined.

Program execution stops when breakpoint #500 is hit (for example, the value of i changes), but execution will continue after the other two breakpoints because they contain go ("g") commands.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsWindbg

Setting File Permissions

PSS ID Number: Q98952

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

In Windows NT, local access controls can be set on just NTFS partitions, not FAT or HPFS partitions or floppies. Read/execute-only permissions should work properly on a CD-ROM.

The exception is that ACLs (access control lists) can be set on shares, regardless of the file system, to control access to all the files within that share. For example, you can give read access to everyone, but give full access just to members of a certain group or to certain individuals.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Setting the Console Configuration

PSS ID Number: Q105674

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

To create a command prompt with custom features such as

```
Settings
Fonts
Screen Size and Position
Screen Colors
```

create a new entry in the Program Manager for CMD.EXE (suppose that the description is CUSTOM), choose these items from the CMD system menu, and select Save Configuration in each dialog box. The settings are saved in the registry under

```
HKEY_CURRENT_USER\
          Console\
          custom
```

and are used when starting the CUSTOM command prompt from the Program Manager or when specifying:

```
start "custom"
```

This behavior is really a convenient side effect of

```
start <string>
```

which sets the title in the window title bar. When you create a new console window with the START command, the system looks in the registry and tries to match the title with one of the configurations stored there. If it cannot find it, it defaults to the values stored in:

```
HKEY_CURRENT_USER\
          Console\
      Configuration
```

This functionality can be duplicated in your own applications using the registry application programming interface (API).

For more information, please see the "Registry and Initialization Files" overview and the REGISTRY sample.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Setup Toolkit .INF File Format and Disk Labels

PSS ID Number: Q88141

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0
- Microsoft Win32 SDK, version 3.5

In the Microsoft Setup Toolkit for Windows, the Source Media Descriptions section of the .INF file contains one line for each of the disks you use to install your application. Each of these lines consists of four quoted strings, separated by commas. The second quoted string is the disk label, which you create using the disk-layout utilities. This disk label has nothing to do with the MS-DOS disk label. The disk label in the .INF file comes from the Disk Labels command on the Options menu in DSKLAYT.EXE and is arbitrarily chosen by the developer during the disk-layout process.

The Setup Toolkit only uses this disk label to prompt the user for disks. The Setup Toolkit uses the tag filename in the Source Media Descriptions section to determine if the proper disk has been placed in the drive.

Additional reference words: 3.00 3.10 3.50 MSSetup tool kit
KBCategory: kbtool
KBSubcategory: TlsMss

SHARE.EXE Functionality Built into Windows NT

PSS ID Number: Q101191

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

SUMMARY

=======

The functionality of the MS-DOS SHARE.EXE utility is built into the Windows NT kernel. Any application or application programming interface (API) that relies on the SHARE.EXE functionality is automatically supported. This functionality cannot be disabled.

MORE INFORMATION

================

If you run the MS-DOS version of SHARE.EXE, you will receive a message stating that SHARE is already installed. The Windows NT MS-DOS emulation hooks Interrupt 2Fh function 10H and always returns a status indicating that SHARE is installed.

If you run an MS-DOS-based application and it complains that SHARE.EXE is not installed, the application may be searching the AUTOEXEC.BAT file for a "share" string rather than using the proper Interrupt 2Fh interface.

Additional reference words: 3.10 3.50

KBCategory: kbprg

KBSubcategory: SubSys

Sharing All Data in a DLL

PSS ID Number: Q109619

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

Win32 dynamic-link libraries (DLLs) use instance data by default. This means that each application that uses a DLL gets its own copy of the DLL's data. However, it is possible to share the DLL data among all applications that use the DLL.

If you only need to share some of the DLL data, we recommend creating a new section and sharing it instead. For additional information, please see the following article(s) in the Microsoft Knowledge Base:

   ARTICLE-ID: Q89817
   TITLE   : How to Specify Shared and Nonshared Data in a DLL

If you want to share *all* of the DLL static data, it is important to do two things:

- First, the DLL must use the DLL version of the C Run-time (for example CRTDLL.LIB or MSVCRT.LIB). Please see your product documentation for more information about using the C Run-time in a DLL.

  NOTE: CRTDLL.LIB is no longer part of the SDK, starting with Windows NT 3.51. It was last released on the April 1995 MSDN 3.5 SDK. Win32 now requires users to specify their own version of C run-time LIBs supplied by their own compiler vender.

- Second, you will need to specify that both .data and .bss are shared. Often, this is done in the "SECTIONS" portion of the .DEF file. For example:

```
SECTIONS
   .bss  READ WRITE SHARED
   .data READ WRITE SHARED
```

  If you are using Visual C++ 32-bit Edition, you have to specify this using the -section switch on the linker. For example:

```
link -section:.data,rws -section:.bss,rws
```

Only static data is shared. Memory allocated dynamically with calls to APIs/functions such as GlobalAlloc() or malloc() are still specific to the calling process.

The system tries to load the shared memory block at the same address in each process. However, if the block cannot be loaded into the same memory address, the system maps the shared section into a different memory address. The memory is still shared. Note that the pointers inside the shared section are invalid under this circumstance and cannot be placed in shared sections.

MORE INFORMATION
================

The C Run-time uses global variables. If the CRT is statically linked to the DLL, these variables will be shared among all clients of the DLL and will most likely cause an exception c0000005.

The reason you need to specify both .data and .bss as shared is because they each hold different types of data. The .data section holds initialized data and the .bss section holds the uninitialized data.

One reason for sharing all data in a DLL is to have consistent behavior in the DLL between Win32 (running on Windows NT) and Win32s (running on Windows 3.1). When running on Win32s, a 32-bit DLL shares its data among all of the processes that use the DLL.

Note that it is not necessary to share all data to behave identically between Win32 and Win32s. The DLL can use thread local storage (TLS) on Win32s to store variables as instance data. For additional information, please see the following article(s) in the Microsoft Knowledge Base:

   ARTICLE-ID: Q109620
   TITLE   : Creating Instance Data in a Win32s DLL

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseDll

Sharing Memory Between 32-Bit and 16-Bit Code on Win32s

PSS ID Number: Q105762

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

SUMMARY

This article discusses many of the issues involved in sharing memory across the process boundary under Win32s.

MORE INFORMATION

Memory allocated by a Win32-based application using GlobalAlloc() can be shared with a 16-bit Windows-based application on Win32s. If the memory is allocated with GMEM_MOVEABLE, then GlobalAlloc() returns a handle and not a pointer. The 16-bit Windows-based application can use the low word of this handle. The high word is all zeros. Make sure to lock the handle using GlobalLock() in the 16-bit Windows-based application to get a pointer.

NOTE: GlobalAlloc (GMEM_FIXED...) is not the same as GlobalFix(GlobalAlloc(GMEM_MOVEABLE...)). GMEM_FIXED will allocate locked pages, which is most often not what you want.

Memory allocated by a 16-bit Windows-based application via GlobalAlloc() must be fixed via GlobalFix() and translated before it can be passed to a Win32-based application. Whenever a Windows object is passed to a Win32-based application by its 32-bit address, the memory must be fixed, because the address is computed from the selector base only once. If Windows moves the memory, the linear address used by the Win32-based application will no longer be valid.

If you are using the Universal Thunk, you can also pass a buffer from a Win32-based application to a 16-bit dynamic-link library (DLL) in the UTRegister() call. The address is translated for you. Another alternative is the translation list passed to the callable stubs. Addresses passed in the translation list will be translated during the thunking process. For more information on the Universal Thunk, please see the "Win32 Programmer's Reference."

NOTE: The ability to share global memory handles under Win32s is a result of the implementation of Windows 3.1, in which all applications run in the same address space. This is not true of existing Win32 platforms and will not be true of future Win32 platforms.

Allocating memory with GlobalAlloc() gets you tiled selectors. However, you can only tile 255 selectors at a time and there is an overall limit of 8192 selectors in the system. If you allocate memory using new, malloc(), HeapAlloc(), LocalAlloc() or VirtualAlloc, your allocated memory does not automatically get you tiled selectors. However, because you don't

automatically get tiled selectors, whenever you pass memory to 16-bit code, selectors must be synthesized for you. There's currently a limit of 256 selectors that Win32s maintains for select synthesis. Also note that each block of memory that you pass is limited to 32K in size due to the way that Win32s tiles selectors.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Sharing Objects with a Service

PSS ID Number: Q106387

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

To share objects (file mapping, synchronization, and so forth) created by a service, you must place a null DACL (discretionary access-control list) in the security descriptor field when the object is created. This grants everyone access to the object.

MORE INFORMATION

================

This null DACL is not the same as a NULL, which is used to specify the default security descriptor. For example, the following code can be used to create a mutex with a null DACL:

```
PSECURITY_DESCRIPTOR    pSD;
SECURITY_ATTRIBUTES     sa;

pSD = (PSECURITY_DESCRIPTOR) LocalAlloc( LPTR,
               SECURITY_DESCRIPTOR_MIN_LENGTH);

if (pSD == NULL)
{
   Error(...);
}

if (!InitializeSecurityDescriptor(pSD, SECURITY_DESCRIPTOR_REVISION))
{
   Error(...);
}

// Add a NULL DACL to the security descriptor..

if (!SetSecurityDescriptorDacl(pSD, TRUE, (PACL) NULL, FALSE))
{
   Error(...);
}

sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = pSD;
sa.bInheritHandle = TRUE;

mutex = CreateMutex( &sa, FALSE, "SOMENAME" );
```

If you are creating one of these objects in an application and the object will be shared with a service, you could also use a null DACL to grant everyone access. As an alternative, you could add an access-control entry (ACE) to the DACL that grants access to the user account that the service is running under. This would restrict access to the object to the service.

For a more detailed example, please see the SERVICES sample.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Sharing Win32 Services

PSS ID Number: Q91698

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

Win32 services are discussed in the overview for the Service Control Manager. The documentation says that:

> A Win32 service runs in a Win32 process which it may or may not share with other Win32 services.

Whether or not a service has its own process is determined by which of these service types is specified in the call to CreateService() to add the service to the Service Control Manager Database.

SERVICE_WIN32_OWN_PROCESS

> This service type indicates that only one service can run in the process. This allows an application to spawn multiple copies of a service under different names, each of which gets its own process. This is the most common type of service.

SERVICE_WIN32_SHARE_PROCESS

> This service type indicates that more than one service can be run in a single process. When the second service is started, it is started as a thread in the existing process. A new process is not created. An example of this is the LAN Manage Workstation and the LAN Manager Server. Note that the service must be started in the system account, which is .\System. The name must be NULL.

The service type for each service is stored in the registry. The values are as follows:

> SERVICE_WIN32_OWN_PROCESS 0x10
> SERVICE_WIN32_SHARE_PROCESS 0x20.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseService

Showing the Beginning of an Edit Control after EM_SETSEL

PSS ID Number: Q64758

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

In a single-line edit control created with the ES_AUTOHSCROLL style, when the EM_SETSEL message is used to select the entire contents of the control, the text in the control is scrolled to the left and the caret is placed at the end of the text. This occurs when the control contains more text than can be displayed at one time. The order of the starting and ending positions specified in the lParam of the EM_SETSEL message makes no difference.

If your application needs to have the entire contents selected and the beginning of the string in view, create the edit control using the ES_MULTILINE style. The order of the starting and ending positions in the EM_SETSEL message is respected by multiline edit controls.

MORE INFORMATION

================

Consider the following example, which sets and then selects the text in a single-line edit control created with the ES_AUTOHSCROLL style:

```
   //hEdit and szText defined elsewhere
   SetWindowText(hEdit, szText);
   SendMessage(hEdit, EM_SETSEL, 0, MAKELONG(0x7FFF, 0));
```

According to the documentation for the EM_SETSEL message, the low-order word of lParam specifies the starting position of the selection and the high-order word specifies the ending position. However, a single-line edit control ignores this ordering and always selects the text from the lower position to the higher position.

If the content of the edit control is longer than the control can display, the text is scrolled to the end of the selection, and the caret is positioned there. In some situations, it is necessary to show the beginning of the text after the selection is made with EM_SETSEL. In Windows 3.00, there is no documented method to accomplish this positioning using a single-line edit control.

A multiline edit control, sized to display only one line and created without the ES_AUTOVSCROLL style, will appear to the user as a

single-line control. However, this control will respect the order of the start and end positions in the EM_SETSEL message.

In the sample code above, a multiline edit control will select the text from the specified starting position to the specified ending position, regardless of which position is higher. In this example, the text is scrolled to the beginning and the caret is placed there. The beginning of the selected text is visible in the control.

NOTE: A multiline edit control uses up slightly more memory in the USER heap than a single-line edit control.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Simulating CreatePatternBrush() on a High-Res Printer

PSS ID Number: Q74793

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

=======

When a pattern brush is used to fill an area of the page on the printer, the printer's high resolution will cause a fine pattern to lose definition and appear as a shade of gray.

Brushes that are created with the CreatePatternBrush() function are 8 pixels by 8 pixels (8 x 8 pixels) in size. On a 300 dots-per-inch (dpi) laser printer, the pattern will be 0.027 inches wide.

To create a pattern that gives similar effects on the screen as on the printer, it is necessary to compare the screen resolution to the printer resolution, and to compensate for the differences.

For example, if the video display is 100 dpi (typical of a VGA), and the printer is 300 dpi (a typical laser printer), the bit must be three times larger in each direction. The following compares a screen bitmap and a printer bitmap:

```
   10101010          111000111000111000111000
   01010101          111000111000111000111000
   10101010          111000111000111000111000
   01010101          000111000111000111000111
   10101010          000111000111000111000111     and so forth
   01010101          000111000111000111000111
   10101010          111000111000111000111000
   01010101          111000111000111000111000
                     111000111000111000111000
    Video            000111000111000111000111
                     000111000111000111000111
                     000111000111000111000111
                     111000111000111000111000
                     111000111000111000111000
                     111000111000111000111000
                     000111000111000111000111
                     000111000111000111000111
                     000111000111000111000111
                     111000111000111000111000
                     111000111000111000111000
                     111000111000111000111000
```

However, since the pattern brush is always 8 x 8 pixels, a different approach must be used when printing:

1. Use the StretchBlt() function to create, from the video bitmap, the 24 x 24 pixel bitmap for the printer.

2. Manually "tile" this bitmap into the region to be painted.

Additional reference words: 3.00 3.10 4.00 95
KBCategory: kbprint
KBSubcategory: GdiPrn

Size Comparison of 32-Bit and 16-Bit x86 Applications

PSS ID Number: Q97765

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

It is expected that a 32-bit version of an x86 application (console or GUI) will be larger than the 16-bit version. Much of this difference is due to the flat memory-model addressing of Windows NT. For each instruction, note that the opcodes have not changed in size, but the addresses have been widened to 32 bits.

In addition, the EXE format under Windows NT (the PE format) is optimized for paging; EXEs are demand-loaded and totally mappable. This leads to some internal fragmentation because protection boundaries must fall on sector boundaries within the EXE file.

The MIPS (or any RISC) version of a Win32-based application typically will be larger and require more memory than its x86 counterpart.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsMisc

SizeofResource() Rounds to Alignment Size

PSS ID Number: Q57808

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SizeofResource() returns the resource size rounded up to the alignment size. Therefore, if you have your own resource types, you cannot use SizeofResource() to get the actual resource byte count.

It has been suggested that this be changed to reflect the actual number of bytes in the resource so that applications can use SizeofResource() to determine the size of each resource. This suggestion is under review and will be considered for inclusion in a future release of the Windows Software Development Kit (SDK).

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrRsc

Small Font Substitution Not Done in Windows NT or Windows 95

PSS ID Number: Q139004

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In Windows versions 3.1 and 3.11, the operating system substituted raster fonts for TrueType fonts when the requested font size fell below approximately 8 points. Typically, the "Small Fonts" raster font would be the substituted font. This font was designed to replace TrueType fonts at very small point sizes because it was more readable and faster to display than most TrueType fonts were at the same sizes.

MORE INFORMATION

Windows 95 and Windows NT no longer do raster font substitution at small point sizes. This is because TrueType fonts that are requested at very small point sizes are now reliably realized without using raster font substitution.

Some 16-bit Windows-based applications depend on the exact extent of strings output with a given font. These applications may not be aware of the fact that fonts realized at small point sizes were automatically substituted with the "Small Fonts" raster font. As a result, these applications may have trouble displaying their output exactly as intended when running under Windows 95 or Windows NT.

Additional reference words: 4.00 smallfont outlinethreshold
KBCategory: kbgraphic kbdisplay
KBSubcategory: GdiFnt GdiTt

SNMP Agent Breaks Up Variable Bindings List

PSS ID Number: Q127870

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

When the SNMP agent receives a request for mutiple variables in a single packet, then for each entry in the variable bindings list, the agent queries the required sub-agent (in this case the .DLL acting as the agent) and packs up the results in a response variable bindings list and returns it in a single packet.

For example, say the varibles requested are:

```
   ip.ipInReceives            (Internet MIB II )
   tcp.tcpMaxConn            (Internet MIB II )

   .iso.org.dod.internet.private.enterprises.lanmanager.lanmgr-2.common.
   comVersionMaj            (LanManager MIB II)

   icmp.icmpOutErrors         (Internet MIB II )
```

In this case, the agent queries the INETMIB2.DLL file twice, the LMMIB2.DLL once, and the INETMIB2.DLL once. Then it packs the results in a response packet and sends it to the requesting manager. There is no "snapshot" of the MIB.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork
KBSubcategory: NtwkSnmp

Sockets Applications on Microsoft Windows Platforms

PSS ID Number: Q124876

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.1, 3.5, 3.51, and 4.0
- Microsoft Win32s version 1.2

SUMMARY
=======

This article documents the resources necessary to do Winsock development on the different Microsoft Windows platforms. The key components necessary for Winsock programming are:

- TCP/IP networking support
- A Windows Sockets include file
- A Windows Wockets import library
- The Windows Sockets Architecture specification.

NOTE: Some implementations of Windows Sockets may support additional protocols, and TCP/IP will not be strictly necessary. See the documentation from your vendor for more information.

MORE INFORMATION
================

Where you get the necessary components for Winsock programming depends on what platform you are using.

Microsoft Windows NT Versions 3.1 and 3.5 and Windows 95

Header filename: WINSOCK.H
Import library name: WSOCK32.LIB

If you are using Microsoft Windows NT version 3.1 or 3.5, the TCP/IP protocol is provided as a component of the operating system. Please see your operating system documentation for more information about installing TCP/IP support for Microsoft Windows NT.

The Winsock header file, import library, and specification are all supplied as part of the Win32 SDK. If you do not have the Win32 SDK, it can be purchased as part of the Microsoft Developer Network Level 2 subscription.

The header file and import library are also supplied with the 32-bit editions of Visual C++. Visual C++ does not include the Windows Sockets specification.

Microsoft Windows for Workgroups Version 3.11

Header filename: WINSOCK.H
Import library name: WINSOCK.LIB

Windows for Workgroups does not include support for the TCP/IP protocol. However, Microsoft does provide a TCP/IP protocol for Windows for Workgroups free of charge. To learn how to obtain the TCP/IP package for Windows for Workgroups, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q111682
   TITLE   : WFWG 3.11: How to Obtain Microsoft DLC and Microsoft TCP/IP

The Winsock include file, import library, and specification are available from the Microsoft Software library. Download WSA16.EXE, a self-extracting file, from the Microsoft Library (MSL) on the following services:

 - Microsoft Download Service (MSDL)
      Dial (206) 936-6735 to connect to MSDL
      Download WSA16.EXE

 - Internet (anonymous FTP)
      ftp ftp.microsoft.com
      Change to the /softlib/mslfiles directory
      get WSA16.EXE

The header file and library for Winsock programming are not included with Visual C++ for Windows.

Microsoft Windows Versions 3.0, 3.1, and 3.11

Header filename: Probably WINSOCK.H (determined by vendor)
Import library name: Probably WINSOCK.LIB (determined by vendor)

TCP/IP support on these versions of Microsoft Windows will have to come from the underlying network. Your network vendor can tell you what TCP/IP support is available.

Depending on your network and the TCP/IP implementation, you may also need to get the Winsock header and library files from your vendor. However, while not guaranteed, the files supplied in WSA16.EXE (see above) may work on your implementation.

Microsoft Win32s Version 1.2

Header filename: WINSOCK.H
Import library name: WSOCK32.LIB

A Winsock application running on Win32s will use the networking support of the underlying system. See the appropriate section above for information about TCP/IP support on the host platform.

The header file, import library, and specification will be part of the

Win32 SDK from which the Win32s application was created. See the section on Windows NT above for more information.

Additional reference words: 3.10 3.50 4.00 1.20 tcpip
KBCategory: kbnetwork kbfile kbnetwork
KBSubcategory: NtwkWinsock W32s

Some Basic Concepts of a Message-Passing Architecture

PSS ID Number: Q74476

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The following is excerpted from an article in the April 1991 issue of Software Design (Japan).

MORE INFORMATION

Asynchronous message-passing means that Windows will send an application messages to act on and that these messages may come in any order. At present, all messages are sent to specific windows. Every window has a function that Windows calls to send that window a message. This function processes the message and returns to Windows. When the function returns to Windows, Windows may then send messages to other windows in the same program, other programs, or to the same window again.

Because most of these messages are generated by user actions (picking an item in a menu, moving a window, and so forth), the specific messages a window receives will differ each time the program is run. This is what makes the messages asynchronous.

This message passing is what makes Windows programming difficult for many programmers. The programmer is no longer writing a program in which he or she controls the flow from beginning to end. Rather, a Windows program is written as a large number of objects, each one designed to handle a specific message from Windows.

Understanding message passing is critical and because it leads to so much confusion, the concept will be explained in greater detail in this article. If message passing is understood, the remainder of Windows can be learned fairly easily. However, it is very unlikely that a Windows program or other graphical user interface (GUI) program can be successfully developed without a thorough understanding of message passing.

In a message-passing system, the focus changes from being proactive (the programmer controls the program flow) to being reactive (Windows controls the program flow). [Or as it has been put by some Macintosh programmers, "Don't call us, we'll call you."] For example, consider the situation where a user chooses an action from a menu in a program. In a proactive program, the program reads the keyboard, determines that the key(s) pressed are

meant to run the action, and calls the function that performs that action. In a reactive system, the program is sent a message indicating that the user chose that item from a menu. When the program receives the message, it calls the function that performs the action. When this function is done, control returns to the system. Although the reactive approach is substantially different from the proactive approach, it is also simpler.

In Windows, every window (including dialog boxes) has a "response function" registered to it. When Windows sends a message to a window, it calls the response function for that window and passes it the message. All messages from Windows are passed to window response functions; there is no other way for Windows to send a message to a program. Therefore, all messages are for a specific window or group of windows.

However, there are four considerations involved with this method:

1. Messages are sent in two distinct ways. The first method consists of messages that are posted to a first-in, first-out queue (PostMessage). The second method consists of messages that are sent (SendMessage). Posted messages, aside from PAINT messages, are serialized, meaning that messages cannot be posted anywhere except to the end of the queue, and the application is sent messages only from the beginning of the queue. Posted PAINT messages are an exception. They are added together and sent only when there is nothing else in the queue. This is done to reduce the number of times a window has to paint itself.

   Messages that are sent are passed to the application immediately, and the send function does not return until the message is processed. However, when a message is posted, an indeterminate number of messages and amount of time will pass before the message is actually sent to a window and acted on. Also, when a message is sent, an message posted earlier may not yet have been acted on.

2. Sending messages or calling functions (which may, as part of their actions, also send messages), can lead to additional messages being generated. The most dangerous situation is where the action for a message generates the same message again. If, while processing a message, an application sends the same message to itself, the application will run out of stack space quite quickly. If an application POSTS the same message to itself, the application will not run out of stack but it will generate an unending stream of messages.

3. If, while processing a message, an application calls a function that sends a message, the application will process the second message in the middle of processing the first message. Therefore, each window response function MUST be fully re-entrant. It is even possible for a function to be re-entered to process the same message as the message currently being processed. For this reason, using global or static variables in a response function is very dangerous.

   Also, if the application uses properties, scratch files and/or other data storage mechanisms, extreme caution is required.

Consider the situation where one message reads in data from a file, then a second message reads in the same data, makes changes to that data, and writes it back to the file. If the first message makes additional changes to the data and writes its new data back to the file, the changes caused by the second message are completely lost. With files, each application must implement its own sharing mechanism. For properties, allocated memory and other memory storage, there is a simple solution: lock the item and use the pointer returned. Because only one lock is allowed, this prevents contention. Never copy the data into a scratch buffer to copy back later.

4. Because messages come in due to user interaction, the application cannot be written to assume that when a particular message is received that another message has already been processed and performed its functions. While, for a given action, a specific sequence of messages may occur, in the interest of remaining completely compatible with potential changes in future versions of Windows, it is recommended that no message ordering dependencies be introduced unless absolutely necessary. The best example of this is that the first message a window gets when it is being created is not WM_CREATE, rather it is WM_GETMINMAXINFO. When one message must logically follow a second (such as WM_CREATE always preceding WM_DESTROY), then it is fine to depend on the specific ordering of those two messages.

The asynchronous, reactive nature of Windows programming can cause confusion. Because the program has no control over the order that messages arrive, the response to ANY specific message CANNOT depend on other messages having been processed or NOT been processed.

To confuse matters even further, an application may be in the middle of processing one message when it calls a Windows function that sends the application another message. When processing this second message, some dependent processing may be only half finished. If an application will check and only do some processing if another message has not already performed it, the application must be prepared for the case where another message has begun the processing, but has not completed it.

Further, when an application sets up a modal dialog box, the DialogBox() function will not return until after the dialog box is dismissed and processing completed. Therefore, after calling the dialog box function, all combinations of user-generated messages may be received before the function returns.

NOTE: 16-bit Windows is non-preemptively multitasks its Windows tasks. Therefore, an application generally does not need to be designed to process a user-originated message in the middle of processing another message. However, when an application calls a Windows function, the application may then get a set of specific messages sent to it by Windows before the called function returns.

32-bit Windows 95 and Windows NT are preemptive multitasking systems. Each thread has its own input queue. It is a good idea to create a separate

thread of execution for the user interface so that it is responsive to user input.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrMsg

Some CTRL Accelerator Keys Conflict with Edit Controls

PSS ID Number: Q67293

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Some keys produce the same ASCII values as CTRL+key combinations. These keys conflict with edit controls if one of the CTRL+key combinations is used as a keyboard accelerator.

The following table lists some of the conflicting keys.

ASCII Value	Key Combination	Equivalent	Windows Virtual Key
0x08	CTRL+H	BACKSPACE	VK_BACK
0x09	CTRL+I	TAB	VK_TAB
0x0D	CTRL+M	RETURN	VK_RETURN

For example, consider the following scenario:

1. CTRL+H has been assigned as an accelerator keystroke to invoke Help

2. An edit control has the focus

3. BACKSPACE is pressed to erase the previous character in the edit control

This results in Help being invoked because pressing BACKSPACE is equivalent to pressing CTRL+H. The edit control does not receive the BACKSPACE key press that it requires because TranslateAccelerator() encounters the 0x08 ASCII value and invokes the action assigned to that accelerator. This limitation is caused by the use of the ASCII key code for accelerators instead of the system-dependent virtual key code.

MORE INFORMATION

When messages for the edit control are processed in a message loop that translates accelerators, this translation conflict will occur. Child windows and modeless dialog boxes are the most common situations where this happens.

The affected keystrokes are translated during the processing of the WM_KEYDOWN message for the letter. For example, when the user types CTRL+H, a WM_KEYDOWN is processed for the CTRL key, then another WM_KEYDOWN is processed for the letter "H". In response to this message, TranslateAccelerator() posts a WM_COMMAND message to the owner of the CTRL+H accelerator. Similarly, when the user presses the BACKSPACE key, a WM_KEYDOWN is generated with VK_BACK as the key code. Because the ASCII value of BACKSPACE is the same as that for CTRL+H, TranslateAccelerator() treats them as the same character. Either sequence will cause a WM_COMMAND message to be sent to the owner of the CTRL+H accelerator, which deprives the child window with the input focus of the BACKSPACE key message.

Because this conflict is inherent to ASCII, the safest way to avoid the difficulty is to avoid using the conflicting sequences as accelerators. Any other ways around the problem may be version dependent rather than a permanent fix.

A second way around the situation is to subclass each edit control that is affected. In the subclass procedure, watch for the desired key sequence(s). The following code sample demonstrates this procedure:

```
/* This code subclasses a child window edit control to allow it to
 * process the RETURN and BACKSPACE keys without interfering with the
 * parent window's reception of WM_COMMAND messages for its CTRL+H
 * and CTRL+M accelerator keys.
 */

/* forward declaration */
long FAR PASCAL NewEditProc(HWND, unsigned, WORD, LONG);

/* required global variables */
FARPROC lpfnOldEditProc;
HWND    hWndOwner;


/* edit control creation in MainWndProc */

TEXTMETRIC tm;
HDC hDC;
HWND hWndEdit;
FARPROC lpProcEdit;

...

 case WM_CREATE:

    hDC = GetDC(hWnd);
    GetTextMetrics(hDC, &tm);
    ReleaseDC(hWnd, hDC);

    hWndEdit = CreateWindow("Edit", NULL,
        WS_CHILD | WS_VISIBLE | ES_LEFT | WS_BORDER,
        50, 50, 50 * tm.tmAveCharWidth, 1.5 * tm.tmHeight,
        hWnd, 1, hInst, NULL);
```

```
    lpfnOldEditProc = (FARPROC) GetWindowLong (hWndEdit, GWL_WNDPROC);
    lpProcEdit = MakeProcInstance ((FARPROC) NewEditProc, hInst);
    SetWindowLong(hWndEdit, GWL_WNDPROC, (LONG) lpProcEdit);
    break;

...

/* subclass procedure */

long FAR PASCAL NewEditProc(HWND hWndEditCtrl, unsigned iMessage,
                            WORD wParam, LONG lParam )
{
  MSG  msg;

  switch (iMessage)
    {
  case WM_KEYDOWN:
    switch (wParam)
        {
    case VK_BACK:
        // This assumes that the next message in the queue will be a
        // WM_COMMAND for the window which owns the accelerators. If
        // this edit control were in a modeless dialog box, hWndOwner
        // should be set to NULL. It may also be NULL in this case.
        PeekMessage(&msg, hWndOwner, 0, 0, PM_REMOVE);

        // Since TranslateAccelerator() processed this message as an
        // accelerator, a WM_CHAR message must be supplied manually to
        // the edit control.
        SendMessage(hWndEditCtrl, WM_CHAR, wParam, MAKELONG(1, 14));
        return 0L;

    case VK_RETURN:
        // Same procedures here.
        PeekMessage(&msg, hWndOwner, 0, 0, PM_REMOVE);
        SendMessage(hWndEditCtrl, WM_CHAR, wParam, MAKELONG(1, 28));
        return 0L;
        }
        break;
    }
    return CallWindowProc(lpfnOldEditProc, hWndEditCtrl, iMessage,
                          wParam, lParam);
}
```

NOTE: Be sure to export the subclass function in the DEF file.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui kbcode
KBSubcategory: UsrCtl

Some Subsystems Persevere Through Logons

PSS ID Number: Q94995

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

Subsystems such as the Local Security Authority (LSA), the spooler, the Win32 server, and the OS/2 subsystem do not terminate when the user logs off, and then restart at next logon.

Windows on Win32 (WOW) is not a true subsystem; rather, it is a user-mode program. When you run an MS-DOS program, the console you run it from has a virtual DOS machine (VDM) attached to it until the console exits (which it is forced to do at logoff). When you run the first Win16 program, the WOW VDM is started and continues to run until the user logs off.

Additional reference words: 3.10 and 3.50
KBCategory: kbprg
KBSubcategory: Subsys

Source-level Debugging Under NTSD

PSS ID Number: Q99053

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

The following are the steps used for source-level debugging under NTSD:

1. Compile using -Zi and -Od.

2. Link using debug:full and debugtype:coff.

3. Load the program into the debugger.

4. Use s+ to change to source mode.

   -or-

   Use s& to change to mixed mode.

For a console application, type "g main" to get to the program start. For a GUI application, type "g WinMain" to get to the program start.

Type "v .<number>" to list source lines starting at <number>. For example, type the following

   v .20

to see all lines starting at line 20.

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsMisc

Specifying Filenames Under the POSIX Subsystem

PSS ID Number: Q99361

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

When specifying filenames under POSIX, use the format

```
   //C/subdir/executable.exe
```

to specify C:\SUBDIR\EXECUTABLE.EXE. If you fail to use this format, you will receive ENAMETOOLONG as the errno.

NOTE: The filenames are case-sensitive.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: Subsys

Specifying Serial Ports Larger than COM9

PSS ID Number: Q115831

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

CreateFile() can be used to get a handle to a serial port. The "Win32 Programmer's Reference" entry for "CreateFile()" mentions that the share mode must be 0, the create parameter must be OPEN_EXISTING, and the template must be NULL.

CreateFile() is successful when you use "COM1" through "COM9" for the name of the file; however, the message "INVALID_HANDLE_VALUE" is returned if you use "COM10" or greater.

If the name of the port is \\.\COM10, the correct way to specify the serial port in a call to CreateFile() is as follows:

```
   CreateFile(
      "\\\\.\\COM10",     // address of name of the communications device
      fdwAccess,          // access (read-write) mode
      0,                  // share mode
      NULL,               // address of security descriptor
      OPEN_EXISTING,      // how to create
      0,                  // file attributes
      NULL                // handle of file with attributes to copy
   );
```

NOTES: This syntax also works for ports COM1 through COM9. Certain boards will let you choose the port names yourself. This syntax works for those names as well.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseCommapi

Specifying Time to Display and Remove a Dialog Box

PSS ID Number: Q74888

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

It is possible to modify the timing of the display of a dialog box. For example, an application has its copyright message in a dialog box that does not have any push buttons. This dialog box is designed to be displayed for five seconds and then to disappear. This article discusses a method to implement this functionality.

MORE INFORMATION

Windows draws the dialog box on the screen during the processing of a WM_PAINT message. Because all other messages (except for WM_TIMER messages) are processed before WM_PAINT messages, there may be some delay before the dialog box is painted. This delay may be avoided by placing the following code in the processing of the WM_INITDIALOG message:

```
ShowWindow(hDlg);
UpdateWindow(hDlg);
```

This code causes Windows to send a WM_PAINT message to the dialog box, bypassing the other messages that may be in the application's queue.

To keep the dialog box on the screen for a particular period of time, a timer should be created during the processing of the WM_INITDIALOG message. When the WM_TIMER message is received, call EndDialog() to close the dialog box.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Specifying Windows "Bounding Box" Coordinates

PSS ID Number: Q27585

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

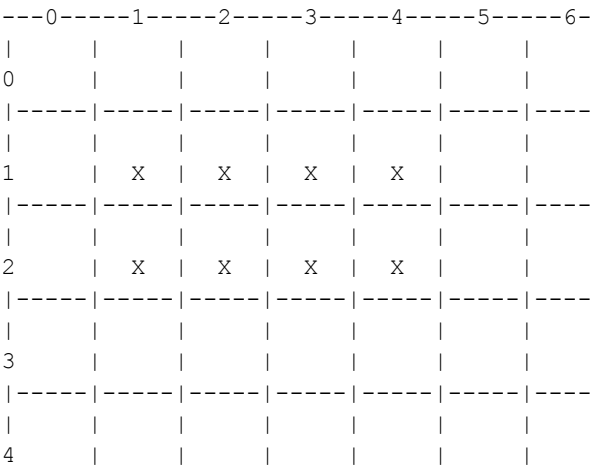
GDI functions, such as Rectangle, Ellipse, RoundRect, Chord, and Pie, have parameters that specify the coordinates of a "bounding box" into which the figure is drawn. Windows draws the figure up to, but not including, the right and bottom coordinates.

MORE INFORMATION

Suppose the following call is made:

```
Rectangle(hDC, 1, 1, 5, 3)
```

Assuming that the device context is using the MM_TEXT mapping mode (in which case logical units map directly to physical pixels), the resulting rectangle will be 4 pixels wide and 2 pixels tall. The following diagram shows which pixels are affected:



It may be helpful to think of the display as a grid, with each pixel contained in a grid cell. The X1, Y1, X2, and Y2 parameters to the

Rectangle function specify an imaginary "bounding box" drawn on the grid. The rectangle is drawn within the bounding box.

The height, width, and area of the resulting rectangle have the following useful properties:

```
   Height = X2 - X1
   Width  = Y2 - Y1
   Area   = Height * Width
```

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiDrw

Starting and Terminating Windows-Based Applications

PSS ID Number: Q105676

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

A 16-bit Windows-based application running under Windows NT version 3.1 is running as a thread in a single virtual MS-DOS machine (VDM). These threads are nonpreemptively scheduled. A 16-bit Windows-based application shares an address space and an input queue with other 16-bit Windows-based applications. Objects created by a thread (application) are owned by the thread (application). This environment is called WOW (Windows on Win32).

Windows NT version 3.5 introduces support for multiple WOWs, so each 16-bit Windows-based application can be run in its own address space. For additional information, please see the following article in the Microsoft Knowledge Base:

ARTICLE-ID: Q115235
TITLE : Running a Windows-Based Application in its Own VDM

When a 16-bit Windows-based application is started by using CreateProcess(), the process handle and the thread handle contained in the PROCESS_INFORMATION structure are pseudo-handles. The only application programming interfaces (APIs) that can use the process handle are WaitForSingleObject(), WaitForMultipleObjects(), and WaitForInputIdle().

NOTE: When you call CreateProcess() with CREATE_SEPARATE_WOW_VDM, the handles returned in the hProcess and hThread fields of the PROCESS_INFORMATION structure are valid handles. The hProcess handle is the handle to the new WOW process, but the hThread handle is the handle to the WOWEXEC thread, and not the thread of the application that you spawned. You can wait on either of these handles and you will be released when the Windows-based application terminates, because the separate VDM goes away when the last application in it terminates. Be aware that if your Windows-based application spawns another Windows-based application and terminates, the VDM stays around, because the application is run in the same VDM as the application that spawned it.

A common question is "How can I terminate a 16-bit process from a 32-bit process?" However, as implied above, PROCESS_INFORMATION.hProcess cannot be used in TerminateProcess() and PROCESS_INFORMATION.dwThreadId cannot be used in PostThreadMessage().

One way to terminate an individual 16-bit Windows-based application is to enumerate the desktop windows using EnumWindows(), determine which is the correct window, obtain the thread ID with GetWindowThreadProcessId(), and post a WM_QUIT message via PostThreadMessage() to terminate the application.

Another way to terminate an individual 16-bit Windows-based application is

to use the TOOLHELP APIs. Use TaskFirst() and TaskNext() to enumerate the tasks, determine which is the correct task, and call TerminateApp() to kill the application. The Windows SDK sample THSAMPLE demonstrates how this can be done.

Additional reference words: 3.10 3.50 win16
KBCategory: kbprg
KBSubcategory: SubSys

StartPage/EndPage Resets Printer DC Attributes in Windows 95

PSS ID Number: Q125696

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows 95 version 4.0

When you print under Windows version 3.x, the printer device context attributes, including things like mapping modes, current pen, current brush, and so on, are reset to their defaults for the device when the end of a page is reached. The escapes NEWFRAME and NEXTBAND and the API EndPage all cause the printer device context to be reset.

When you print under Windows NT version 3.x, the printer device context attributes are not reset during a print job.

When you print under Windows 95, the point at which the printer device context is reset to the default attributes depends on what version the executable was marked as. For executables marked as 3.x, the printer device context will be reset when EndPage is called. For executables marked as 4.0, the printer device context will be reset when StartPage is called. This applies to both 16-bit-based and Win32-based executables running under Windows 95.

A 16-bit-based executable's version can be set by using the Resource compiler's /xx switch where xx is 30, 31, or 40. A Win32-based executable's version can be set by using the /SUBSYSTEM:windows,x.x linker switch.

Additional reference words: 4.00
KBCategory: kbprint
KBSubcategory: GdiPrn

Stroke Fonts Marked as OEM Character Set Are ANSI

PSS ID Number: Q72020

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

There are three stroke (or vector) fonts packaged with Windows versions 3.0 and 3.1 and Windows NT: Roman, Script, and Modern. These fonts are marked as belonging to the OEM character set when, in fact, they belong to the ANSI character set. NOTE: Windows 95 provides only the Modern vector font. The Roman and Script fonts are included in the True Type fonts shipped with the system.

The OEM character set is the character set used by the hardware device on which Windows is running (for example, the IBM PC). The IBM PC OEM character set and ANSI character set are listed in "Microsoft Windows Software Development Kit Reference Volume 2" for the Windows SDK version 3.0 and in "Programmer's Reference, Volume 3: Messages, Structures, and Macros" for the Windows SDK version 3.1.

The fonts were marked in this manner for two reasons. First, in previous versions of Windows, the stroke fonts did include non-ANSI characters. Second, mismarking the character set ensures proper font mapping. The character-set attribute of a font is assigned a very high penalty weight in the font mapping scheme. If stroke fonts were not marked as using the OEM character set, a stroke font might be chosen by the font mapper [during a SelectObject() call] instead of a raster font when a requested raster font size is not available. This behavior occurs because most raster fonts belong to the ANSI character set, character size has much lower penalty weight than character set, and stroke fonts can be scaled to any desired size. Some raster fonts can be scaled; however, they can be scaled only to specific sizes.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Support for Sleep() on Win32s

PSS ID Number: Q100713

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

The Win32 application programming interface (API) documentation indicates that Sleep() is supported on Win32s. It is important to note, however, that the behavior of Sleep() on Win32s is not the same as it is under Windows NT.

Under Win32s, Sleep() calls Yield(). The Windows version 3.1 Yield() function yields only if the message queue is empty; therefore, Sleep() cannot be relied on to do anything. Use a PeekMessage() loop to do idle time processing.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Switching Between Single and Multiple List Boxes

PSS ID Number: Q57959

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

After creating a list box with the CreateWindow() API, changing the list box from a single selection to a multiple selection can be accomplished in the following way:

Create two hidden list boxes in the .RC file, and during the WM_INITDIALOG routine, display one of the boxes. Change between the two by making one hidden and the other one visible using the ShowWindow function.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Symbolic Information for System DLLs

PSS ID Number: Q103862

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

=======

The debugging information for the system dynamic-link libraries (DLLs) is contained separately in files with a .DBG extension. The Win32 SDK setup, SETUPSDK, will install the following .DBG files by default in <SYSTEMROOT>\SYMBOLS\DLL:

ADVAPI32.DBG	OLECLI32.DBG
COMDLG32.DBG	OLESVR32.DBG
CRTDLL.DBG	RASAPI32.DBG
DLCAPI.DBG	RPCNS4.DBG
GDI32.DBG	RPCRT4.DBG
INETMIB1.DBG	SHELL32.DBG
KERNEL32.DBG	USER32.DBG
LMMIB2.DBG	VDMDBG.DBG
LZ32.DBG	VERSION.DBG
MGMTAPI.DBG	WIN32SPL.DBG
MPR.DBG	WINMM.DBG
NDDEAPI.DBG	WINSTRM.DBG
NETAPI32.DBG	WSOCK32.DBG
NTDLL.DBG	

For Windows NT version 3.5, the additional .DBG files are:

- GLU32.DBG
- MSACM32.DBG
- OLEAUT32.DBG
- OPENGL32.DBG

For Windows NT versions 3.51, the additional .DBG files are:

- COMCTL32.DBG
- DHCPMIB.DBG
- OLE32.DBG
- OLEDLG.DBG
- RICHED32.DBG
- WINSMIB.DBG

Note that these files are not installed by the alternative Win32 SDK install method, MANUAL.BAT. Therefore, WinDbg will warn that symbol information cannot be found for each of the system DLLs called by the debuggee.

These .DBG files can be manually installed by copying them from the SDK CD. For x86, they are located in \SUPPORT\DEBUG\I386\SYMBOLS\DLL. For MIPS, they are located in \SUPPORT\DEBUG\MIPS\SYMBOLS\DLL. Note that there are more than 200 .DBG files in each of these directories.

MORE INFORMATION

In Windows NT 3.1, there are also debugging versions of the system DLLs that can be installed by using SWITCH.BAT, which is located on the CD in \SUPPORT\DEBUGDLL. Refer to page 11 of the "Getting Started" manual and the batch file itself for more information.

Additional reference words: 3.10 3.50
KBCategory: kbsetup
KBSubcategory: Setins

System Versus User Locale Identifiers

PSS ID Number: Q100488

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

In Windows NT version 3.1, the following pairs of application programming interfaces (APIs) have the same functionality:

```
GetSystemDefaultLCID()   and GetUserDefaultLCID()
GetSystemDefaultLangID() and GetUserDefaultLangID()
```

The user LangID and LCID are always set to the system value. In future versions of Windows NT, it will be possible to set the LangID and the LCID on a per-user basis.

Note that it is possible to set the LCID on a per-thread basis [that is, SetThreadLocale()] in Windows NT 3.1.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrLoc WintlDev

SystemParametersInfo() Add-On Gets or Sets System Parameters

PSS ID Number: Q125695

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with Microsoft Windows 95

The SystemParametersInfo() API provides functionality to get or set many Windows System Parameters based on the action flag if Windows extension #1 is installed. Here are some examples:

- The Full Window Drag system parameter can be retrieved or set by using SPI_GETDRAGFULLWINDOWS/SPI_SETDRAGFULLWINDOWS as the action flag in the SystemParametersInfo() API. When full drag windows is enabled, users can move entire windows instead of just moving the outline. This makes it easier to see how a window looks while being resized. This feature is not available with Windows, so calling SystemParametersInfo using the SPI_GETDRAGFULLWINDOWS or SPI_SETDRAGFULLWINDOWS flag will always fail. However, this feature will be available if an add-on product, Windows extension #1, is installed.

- The Font Smoothing setting can be retrieved or set by using SPI_GETFONTSMOOTHING or SPI_SETFONTSMOOTHING as the action flag in the SystemParametersInfo() API. Enabling this system setting tells Windows to draw smooth characters using font anti-aliasing. This feature is not available with Windows. Thus the call to SystemParametersInfo with the SPI_GETFONTSMOOTHING or SPI_SETFONTSMOOTHING flag will always fail. However, this Font Smoothing feature will be available if an add-on product, Windows extension #1, is installed.

- SystemParametersInfo(SPI_GETWINDOWSEXTENSION, 1, 0, 0) returns true if Windows extension #1 is installed otherwise false. The Second parameter denotes Windows extension #1.

Additional reference words: 4.00 95
KBCategory: kbui
KBSubcategory: UsrWnd

Taking Ownership of Registry Keys

PSS ID Number: Q111546

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

To take ownership of a registry key it is necessary to have a handle to the key. A handle to the key can be obtained by opening the key with a registry API (application programming interface) such as RegOpenKeyEx(). If the user does not have access to the registry key, the open operation will fail and this will in turn prevent ownership being taken (because a handle to the key is required to change the key's security).

The solution to this problem is to first enable the TakeOwnership privilege and then to open the registry key with WRITE_OWNER access as shown below:

```
RegOpenKeyEx(HKEY_CLASSES_ROOT,"Testkey",0,WRITE_OWNER,&hKey);
```

This function call will provide a handle to the registry, which can be used in the following call to take ownership:

```
RegSetKeySecurity(hKey,OWNER_SECURITY_INFORMATION, &SecurityDescriptor);
```

Please note that you will need to initialize the security descriptor being passed to RegSetKeySecurity() and set the owner field to the new owner SID.

Taking ownership of a registry key is not a common operation. It is typically an operation that an administrator would use as a last resort to gain access to a registry key.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

TAPI Call Handles Returned from an Asynchronous LINE_REPLY

PSS ID Number: Q132191

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.51, 4.0
- Microsoft Windows Telephony Software Development Kit (TAPI SDK) version 1.0

Several TAPI APIs, such as lineMakeCall and lineSetupConference, take the address of a call handle as a parameter. Upon successful completion of these APIs, the address pointed at is filled with the new call handle.

TAPI does not, however, fill this address until the application receives the LINE_REPLY message indicating that the API completed successfully. Thus, not only is the handle not valid until the LINE_REPLY, but the actual address must still be valid when LINE_REPLY is recieved. It is recommended that stack variables not be used because they will often go out of scope before the LINE_REPLY message is received.

For example, this code fragment is incorrect:

```
func()
{
    HCALL hCall;
...
    AsyncID = lineMakeCall(hLine, &hCall, lpszDestAddress, 0, NULL);
}
```

Here, lineMakeCall is going to return an asynchronous ID that will eventually have a matching LINE_REPLY message. However, by the time the LINE_REPLY message is retrieved, hCall will not be a valid variable and &hCall will not be a valid pointer. Expect a general protection (GP) fault or stack corruption.

In the following code, When LINE_REPLY for lineMakeCall is received, &g_hCall is still valid and is filled with a valid call handle:

```
HCALL g_hCall;
func()
{
...
    lineMakeCall(hLine, &g_hCall, lpszDestAddress, 0, NULL);

    // g_hCall is *not* valid until LINE_REPLY is received
}
```

Additional reference words: 1.00 1.30 1.40 4.00
KBCategory: kbprg
KBSubcategory: TAPI

Text Alignment in Single Line Edit Controls

PSS ID Number: Q108940

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

Text in single line edit controls cannot be centered or right-aligned (right-justified). Text in single line edit controls is left-aligned (left-justified) by default. This is by product design, and therefore specifying the ES_RIGHT or ES_CENTER style while creating the single line edit control does not have any effect on the text alignment.

Windows does not allow text to be centered or right-aligned in a single line edit control. However, an easy way to work around this problem is to create a multiline edit control that is the same size as a single line edit control.

Text in a multiline edit control can be centered or right/left aligned. Note that the multiline edit control should be created without the WS_VSCROLL or ES_AUTOVSCROLL style.

While single line edit controls may support text alignment in a future release of Windows, Windows 3.x applications must use multiline edit controls to achieve the same effect.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

The Clipboard and the WM_RENDERFORMAT Message

PSS ID Number: Q31668

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The clipboard sends a WM_RENDERFORMAT message to an application to request that application format the data last copied to the clipboard in the specified format, and then pass a handle to the formatted data to the clipboard.

If an application cannot supply the requested data, it should return a NULL handle. Because most applications provide access to the actual data (not rendered) through the CF_TEXT format, applications that use the clipboard can get the applicable data when rendering fails.

If the application cannot render the data because the system is out of memory, the application can call GlobalCompact(-1) to discard and compress memory, then try the memory allocation request again.

If this fails to provide enough memory, the application can render the data into a file. However, applications that use this technique must cooperate in order to know that the information is in a file, the name of the file, and the format of the data.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrClp

The Parts of a Windows Combo Box and How They Relate

PSS ID Number: Q65881

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

A Windows combo box is a compound structure composed of individual windows. Three types of windows can be created as part of a combo box:

- A combo box itself, of window class "ComboBox"
- An edit control, of window class "Edit"
- A list box, of window class "ComboLBox"

The relationship among these three windows varies depending upon the different combo box styles.

MORE INFORMATION

For combo boxes created with the CBS_SIMPLE styles, the ComboBox window is the parent of the edit control and the list box that is always displayed on the screen. When GetWindowRect() is called for a combo box of this style, the rectangle returned contains the edit control and the list box.

Combo boxes created with the CBS_DROPDOWNLIST style have no edit control. The region of the combo box that displays the current selection is in the ComboBox window itself. When GetWindowRect() is called for a combo box of this style, the rectangle returned does not include the list box.

For combo boxes created with the CBS_DROPDOWN style, three windows are created. The combo box edit control is a child of the ComboBox window. When GetWindowRect() is called for a combo box of this style, the rectangle returned does not include the list box.

However, the ComboLBox (list box) window for combo boxes that have the CBS_DROPDOWN or CBS_DROPDOWNLIST style is not a child of the ComboBox window. Instead, each ComboLBox window is a child of the desktop window. This is required so that, when the drop-down list box is dropped, it can extend outside the application window or dialog box. Otherwise, the list box would be clipped at the window or dialog box border.

Because the ComboLBox window is not a child of the ComboBox window, there is no simple method to get the handle of one window, given the other. For example, given a handle to the ComboBox, the handle to any associated drop-

down list box is not readily available. The ComboLBox is a private class registered by USER that is a list box with the class style CS_SAVEBITS.

Additional reference words: control focus release 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

The SBS_SIZEBOX Style

PSS ID Number: Q102485

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

A size box is a small rectangle the user can expand to change the size of the window. When you want a size box, you create a SCROLLBAR window with the SBS_SIZEBOX flag. This action creates a size box with the height, width, and position that you specified in the call to CreateWindow. If you specify SBS_SIZEBOXBOTTOMRIGHTALIGN, the box will be aligned in the lower right of the rectangle you specified when creating the window. If you specify SBS_SIZEBOXTOPLEFTALIGN, the box will be aligned in the upper left of the rectangle you specified in your call to CreateWindow().

MORE INFORMATION

The user moves the mouse pointer over to the box, presses and holds the left mouse button, and drags the mouse pointer to resize the window. When the user does this, the borders on the window (the frame) move. When the user releases the mouse button, the window is resized.

You create a size box by creating a child window of type WS_CHILD | WS_VISIBLE | SBS_SIZEBOX | SBS_SIZEBOXTOPLEFTALIGN. You don't have to do any of the processing for this; the system will take care of it. You will notice in your window procedure that you will get the scroll bar messages plus the WM_MINMAXINFO message. Size boxes work similar to the way the WS_THICKFRAME/WS_SIZEBOX style does on a window.

Under Windows NT, applications that create a size box either using WS_SIZEBOX or WS_THICKFRAME or by created the SBS_SIZEBOX control have no way of showing the user that such a feature exists. Under Windows 95, the size box appears as a jagged edge, usually at the bottom right corner.

NOTE: Make sure that the main window is created with the WS_VSCROLL and WS_HSCROLL styles.

Additional reference words: 3.10 3.50 4.00 sizebox
KBCategory: kbui
KBSubcategory: UsrCtl

The Use of PAGE_WRITECOPY

PSS ID Number: Q105532

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

The documentation to CreateFileMapping(), VirtualAlloc(), and VirtualProtect() indicates that the PAGE_WRITECOPY protection gives copy-on-write access to the committed region of pages. As it is, PAGE_WRITECOPY makes sense only in the context of file mapping, where you want to map something from the disk into your view and then modify the view without causing the data to go on the disk.

The only case where VirtualAlloc() should succeed with PAGE_WRITECOPY is the case where CreateFileMapping() is called with -1 and allocates memory with the SEC_RESERVE flag and later on, VirtualAlloc() is used to change this into MEM_COMMIT with a PAGE_WRITECOPY protection.

There is a bug in Windows NT 3.1 such that the following call to VirtualAlloc() will succeed:

```
lpCommit = VirtualAlloc(lpvAddr, cbSize, MEM_COMMIT, PAGE_WRITECOPY);
```

This call will fail under Windows NT 3.5.

NOTE: lpvAddr is a pointer to memory that was allocated with MEM_RESERVE and PAGE_NOACCESS.

One case where this might be useful is when emulating the UNIX fork command. Emulating fork behavior would involve creating instance data and using threads or multiple processes.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMm

The Use of the SetLastErrorEx() API

PSS ID Number: Q97926

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SetLastErrorEx() is intended for better debugging support, not for passing additional error information.

The SetLastErrorEx() application programming interface (API) differs from the SetLastError() API only in that it raises a debug "RIP" event. The RIP event is intended to give text to the debugger so that the user can retry, ignore, and so forth, these errors. SetLastErrorEx() raises an exception only if SetDebugErrorLevel() has been called by the debugger to allow the errors to be passed on.

The error type can be determined from the debugger by examining the debug event structure that is passed with the event. The debug event structure contains a RIP_INFO substructure.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMisc

Thread Handles and Thread IDs

PSS ID Number: Q127992

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The CreateThread() API is used to create threads. The API returns both a thread handle and a thread identifier (ID). The thread handle has full access rights to the thread object created. The thread ID uniquely identifies the thread on the system level while the thread is running. The ID can be recycled after the thread has been terminated. This relationship is similar to that of the process handle and the process ID (PID).

There is no way to get the thread handle from the thread ID. While there is an OpenProcess() API that takes a PID and returns the handle to the process, there is no corresponding OpenThread() that takes a thread ID and returns a thread handle.

The reason that the Win32 API does not make thread handles available this way is that it can cause damage to an application. The APIs that take a thread handle allow suspending/resuming threads, adjusting priority of a thread relative to its process, reading/writing registers, limiting a thread to a set of processors, terminating a thread, and so forth. Performing any one of these operations on a thread without the knowledge of the owning process is dangerous, and may cause the process to fail.

If you will need a thread handle, then you need to request it from the thread creator or the thread itself. Both the creator or the thread will have a handle to the thread and can give it to you using DuplicateHandle(). This requirement allows both applications to coordinate their actions.

NOTE: You can also take full control of the application by calling DebugActiveProcess(). Debuggers receive the thread handles for a process when the threads are created. These handles have THREAD_GET_CONTEXT, THREAD_SET_CONTEXT, and THREAD_SUSPEND_RESUME access to the thread.

Additional reference words: 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseProcThrd

Thread Local Storage Overview

PSS ID Number: Q94804

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Thread local storage (TLS) is a method by which each thread in a given process is given a location(s) in which to store thread-specific data.

Four functions exist for TLS: TlsAlloc(), TlsGetValue(), TlsSetValue(), and TlsFree(). These functions manipulate TLS indexes, which refer to storage areas for each thread in a process. A given index is valid only within the process that allocated it.

Note that the Visual C++ compiler supports an alternate syntax:

```
   _declspec( thread )
```

which can be used in place of directly calling these APIs. Please see the description of _declspec in the VC++ "Language and Run-time Help" helpfile for more information.

MORE INFORMATION
================

A call to TlsAlloc() returns a global TLS index. This one TLS index is valid for every thread within the process that allocated it, and should therefore be saved in a global or static variable.

Thread local storage works as follows: when TlsAlloc() is called, every thread within the process has its own private DWORD-sized space reserved for it (in its stack space, but this is implementation-specific). However, only one TLS index is returned. This single TLS index may be used by each and every thread in the process to refer to the unique space that TlsAlloc() reserved for it.

For this reason, TlsAlloc() is often called only once. This is convenient for DLLs, which can distinguish between DLL_PROCESS_ATTACH (where the first process's thread is connecting to the DLL) and DLL_THREAD_ATTACH (subsequent threads of that process are attaching). For example, the first thread calls TlsAlloc() and stores the TLS index in a global or static variable, and every other thread that attaches to the DLL refers to the global variable to access their local storage space.

Although one TLS index is usually sufficient, a process may have up to TLS_MINIMUM_AVAILABLE indexes (guaranteed to be greater than or equal

to 64).

Once a TLS index has been allocated (and stored), the threads within the process may use it to set and retrieve values in their storage spaces. A thread may store any DWORD-sized value in its local storage (for example, a DWORD value, a pointer to some dynamically allocated memory, and so forth). The TlsSetValue() and TlsGetValue() APIs are used for this purpose.

A process should free TLS indexes with TlsFree() when it has finished using them. However, if any threads in the process have stored a pointer to dynamically allocated memory within their local storage spaces, it is important to free the memory or retrieve the pointer to it before freeing the TLS index, or it will be lost.

For more information, please see "Using Thread Local Storage" in the "Processes and Threads" overview in the "Win32 Programmer's Reference".

Example

Thread A within a process calls TlsAlloc(), and stores the index returned in the global variable TlsIndex:

```
   TlsIndex = TlsAlloc();
```

Thread A then allocates 100 bytes of dynamic memory, and stores it in its local storage:

```
   TlsSetValue( TlsIndex, malloc(100) );
```

Thread A creates thread B, which stores a handle to a window in its local storage space referred to by TlsIndex.

```
   TlsSetValue( TlsIndex, (LPVOID)hSomeWindow );
```

Note that TlsIndex refers to a different location when thread B uses it, than when thread A uses it. Each thread has its own location referred to by the same value in TlsIndex.

Thread B may terminate safely because it does not need to specifically free the value in its local storage.

Before thread A terminates, however, it must first free the dynamically allocated memory in its local storage

```
   free( TlsGetValue( TlsIndex ) );
```

and then free the TLS index:

```
   if ( !TlsFree( TlsIndex ) )
      // TlsFree() failed.  Handle error.
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseProcThrd

Time Stamps Under the FAT File System

PSS ID Number: Q101186

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

Windows NT considers the time stamp on a file stored on a FAT (file allocation table) partition to be standard time if the current time is standard time, and daylight time if the current time is daylight time, regardless of what time of year the file was originally time stamped.

This is not an issue under NTFS, which consistently implements Universal Coordinated Time.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

Timer Resolution in Windows NT

PSS ID Number: Q115232

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

In Win32-based applications, the GetTickCount() timer resolution on Windows NT, version 3.1, is 15 milliseconds (ms) for x86 and 10 ms for MIPS and Alpha. On Windows NT, version 3.5, the GetTickCount() timer resolution is 10 ms on the 486 or greater, but the resolution is still 15 ms on a 386.

NOTE: The measurements in milliseconds indicate the period of the interrupt, not the units of the returned value.

The Win32 API QueryPerformanceCounter() returns the resolution of a high-resolution performance counter if the hardware supports one. For x86, the resolution is about 0.8 microseconds (0.0008 ms). For MIPS, the resolution is about twice the clock speed of the processor. You need to call QueryPerformanceFrequency() to get the frequency of the high-resolution performance counter.

NOTE: These numbers are likely to change in future versions.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Tips for Debugging ISAPI DLLs

PSS ID Number: Q152054

The information in this article applies to:

- Microsoft Win32 SDK, version 4.0
- Microsoft Internet Information Server, versions 1.0, 2.0

SUMMARY

Writing an ISAPI extension or filter is no more difficult than writing any other DLL, but debugging the DLL can be a challenge. This article provides a few ideas for debugging an ISAPI DLL.

The following acronyms are used in this article:

ISAPI	- Internet Server Application Programming Interface
WWW	- World Wide Web
IIS	- Internet Information Server
SDK	- Software Development Kit
DLL	- Dynamic Link Library
HTML	- Hypertext Markup Language

The following approaches are discussed:

- Turning off DLL caching
- Using OutputDebugString in an ISAPI DLL
- Using MessageBox
- Writing output to a log file
- Running IIS interactively
- Using an alternate heap for memory
- Using CGIWRAP to debug without a server
- Using ISmoke to test an extension

MORE INFORMATION

ISAPI is a subset of the Win32 API. It allows WWW servers, such as Microsoft's IIS, to be extended in two ways: extensions and filters. In an ISAPI extension, a browser typically sends information entered in a form, and the extension returns a complete HTML page built programmatically. In an ISAPI filter, all browser data, both inbound and outbound, can be modified before or after the server processes it. Extension and filter DLLs must conform to a specification; there are specific functions that must be implemented in order for the DLL to work.

Microsoft's IIS supports ISAPI on Windows NT Advanced Server versions 3.51 and 4.0. Microsoft's Peer Web Server supports ISAPI on Windows NT Workstation 4.0 and Windows 95. This article focuses on IIS for Windows NT Advanced Server, but many of the tips apply to all implementations of ISAPI, even Web servers other than Microsoft's.

Tips for ISAPI Development

IIS runs as a Windows NT service in the local system account, which is what makes debugging ISAPI DLLs difficult. Running as a service introduces issues that are new to many programmers:

- The DLL runs in the local system context.
- There is no desktop.
- ISAPI extension procedures have an impersonated security context.
- ISAPI developers do not have access to the server source code so the developers are extending an application they did not write.

The following tips are intended to aid in the development of ISAPI DLLs.

Turning Off DLL Caching

It is often desirable during development to force your ISAPI DLL to unload after each request, allowing you to replace it with a new version. IIS, by default, will hold the DLL in memory, keeping the DLL file in use. To modify this default cache setting, use the registry editor to change the following key:

```
HKEY_LOCAL_MACHINE
 System
  CurrentControlSet
   Services
    W3SVC
     Parameters
```

Set CacheExtensions to 0 to disable caching, or set it to 1 to enable caching. When caching is turned off, extra loading and unloading makes ISAPI extensions dramatically slower, so use this option only for development.

Filter DLLs are always loaded when the IIS service is running, and there is no way to change filter loading behavior. You must stop the IIS service with the administration tool, copy your new version over the old, and then restart the WWW service. The administration tool can run on a remote development machine controlling the server over the network.

OutputDebugString

An easy technique to debug either an ISAPI filter or extension is to use OutputDebugString. This function is part of Win32 and it is a standard way to send a string to a debug monitor. Use DBMON, included as a sample in the Win32 SDK, to view debug strings. A fix is necessary to view ISAPI debug output with DBMON, because the DLL runs in the local system account while DBMON runs in the logged-in user account. Starting with the Win32 SDK for Windows NT 4 beta 2, DBMON creates a NULL DACL and passes the security

attributes to both CreateEvent calls and the CreateFileMapping call. You can modify DBMON yourself if you do not have the current SDK. The code below shows how to make a NULL DACL:

```
   SECURITY_ATTRIBUTES sa;    SECURITY_DESCRIPTOR sd;

   sa.nLength = sizeof(SECURITY_ATTRIBUTES);
   sa.bInheritHandle = TRUE;
   sa.lpSecurityDescriptor = &sd;

   if(!InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION))
      return;   // Handle errors
   if(!SetSecurityDescriptorDacl(&sd, TRUE, (PACL)NULL, FALSE))
      return;   // Handle errors

   // You may now pass &sa to CreateEvent and CreateFileMapping
```

MessageBox

The Online Help for MessageBox in the Win32 SDK describes a number of rarely-used constants, including MB_SERVICE_NOTIFICATION and MB_TOPMOST. In the context of ISAPI, add both of these flags to your message boxes because without them, the message box call will fail. Your ISAPI DLL is running as a service so it does not have a desktop, and these flags tell the operating system to display the message on the logged-in user's desktop.

Please note that the value of MB_SERVICE_NOTIFICATION has changed on Windows NT 4.0 because of a conflict with Windows 95. When you specify MB_SERVICE_NOTIFICATION|MB_TOPMOST, your message will display properly on all versions of Windows NT and Windows 95.

Do not use MessageBox for anything but debugging. If you need to record error messages, use the Windows NT event log. See the Win32 sample "logging" for information on how to use the event log. Depending on the SDK version you have, the logging sample may be in the q_a subdirectory off the root of the SDK CD, or it may be with other samples in \mstools\samples.

Output to a Log File

An easy way to track the flow of an ISAPI extension or filter is to use a log file. You can use the normal Win32 file I/O functions such as CreateFile, ReadFile, WriteFile, and CloseHandle. Your DLL's current directory will be the system root (c:\winnt40\system32). Also, keep security in mind because the default security context for your DLL is the local system account. This applies even when impersonation is active. If you do not allow the local system to create files in the appropriate directory and do not specify a non-NULL security attribute, your CreateFile call will fail.

The ISAPI homefilt sample in the SDK provides one example of a file-logging mechanism.

Running IIS Interactively

A popular debugging option is to run IIS as a console application and use Visual C++ or another debugger for full source debugging. Information to do this is given in the SDK readme. Prior to Win32 SDK version 4 beta 2, the readme was included in the ActiveX SDK from http://www.microsoft.com/intdev. As of version 4 beta 2, ISAPI became part of the Win32 SDK. Examine the Win32 SDK readme for the procedure to run IIS as a console application.

Briefly, here are a few tips. IIS cannot run as a console application and a service at the same time. You will have to stop all three IIS services, WWW, FTP, and Gopher, before running IIS from a command prompt. Once IIS is running, you can use PVIEW (part of Visual C++) or TLIST (part of the NT resource kit) to get the process ID of INETINFO.EXE, and then use your debugger to attach to the running process.

Another approach is to supply the full command line, inetinfo.exe -e W3SVC, as described in the SDK readme, within the debug settings. For Visual C++, choose the Build menu, select Settings, then select the Debug tab. Type the full path name of "inetinfo.exe" as the Executable for Debug Session, and type "-e W3SVC" in Program Arguments. See Visual C++ 4.1 tech note TN063: Debugging Internet Extension DLLs, for details on using the Visual C++ debugger.

Running IIS as a console application still is not exactly the same as running it as a service. For instance, authentication filters do not function properly in the console version of IIS. Instead of debugging IIS as a console application, it is possible to debug IIS running as a service. Attach a debugger to the INETINFO.EXE process, then use a browser and interact with your web site. When a debugger is attached to IIS, any occurrence of DebugBreak() will stop the service and put the debugger in full-source debugging mode. Each time you modify your DLL code, detach and restart the service using the Microsoft Internet Service Manager tool.

The main cost of running IIS interactively is the start-up time required. It takes a few seconds to start and stop your debugger and, during development, this may become unacceptable. Be sure to consider the other alternatives.

Using an Alternate Heap for Memory

An ISAPI DLL might share the process heap with the WWW service. ISAPI DLL memory overwrites may not show up until later, and they may appear to be a problem with IIS. When you suspect memory corruption problems, modify your DLL code to use an alternate heap by calling HeapCreate, then HeapAlloc and HeapFree for all memory allocations. If switching from the process heap to a new heap changes the behavior of a crash, you can conclude that your DLL is corrupting memory.

See the ISAPI sample CGIWRAP to isolate your extension DLL from the server. CGIWRAP is an executable and will have its own address space so even when your DLL corrupts memory, IIS will remain safe. Also, the CGIWRAP process is terminated after each request is complete. This technique makes your

extension very slow but is a helpful interim solution until the corruption bug is finally fixed.

The Win32 family of VirtualAlloc functions allows you to allocate 64K blocks of memory and can modify the protection of a block of memory. It can be helpful to use the memory protection provided by these functions to diagnose memory overwrite bugs.

Finally, before your DLL is deployed in a production environment, check for memory leaks. For example, cut and paste the following code to track the number of bytes allocated. Call DumpBytesInUse() within your code to track the number of bytes currently allocated:

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#if 1    // change to #if 0 to turn off mem check

DWORD g_dwTotal;

//
// Call MemAlloc just like you would call HeapAlloc
//

LPVOID MemAlloc (HANDLE hHeap, DWORD dwFlags, DWORD dwSize)
   {
   LPVOID p;

   p = HeapAlloc (hHeap, dwFlags, dwSize);
   if (p)
      g_dwTotal += dwSize;

   return p;
   }

BOOL MemFree (HANDLE hHeap, DWORD dwFlags, LPVOID p)
   {
   DWORD dwSize;
   BOOL bReturn;

   dwSize = HeapSize (hHeap, dwFlags, p);
   bReturn = HeapFree (hHeap, dwFlags, p);

   if (dwSize != 0xffffffff && bReturn)
      g_dwTotal -= dwSize;

   return bReturn;
   }

void DumpBytesInUse (void)
   {
   TCHAR szMsg[256];

   wsprintf (szMsg, "Bytes in use: %u\r\n", g_dwTotal);
   OutputDebugString (szMsg);
```

```
    }

#else

#define MemAlloc HeapAlloc
#define MemFree HeapFree
#define DumpBytesInUse(x)

#endif
```

Using CGIWRAP to Debug Without a Server

An effective way to conduct basic testing on an ISAPI DLL is to use the CGIWRAP sample. CGIWRAP is an executable that calls an ISAPI DLL. Although it is intended to allow an ISAPI DLL run as a CGI executable, CGIWRAP can be modified for use with a debugger.

There are two sources of input for an ISAPI DLL: HTTP header variables and HTTP data. An ISAPI DLL retrieves header variables from GetServerVariable and retrieves data from ReadClient. Similarly, a CGI executable gets its header variables from the environment (with getenv) and gets its data from standard input.

This means you can simulate a WWW server from the Command Line. Set environment variables to match what is given to your ISAPI DLL and pipe a file into CGIWRAP to simulate the inbound form data given to your DLL. When CGIWRAP runs, it will translate environment variables and the piped input into an ISAPI interface. Because you have the source to CGIWRAP, you can set breakpoints on every line of code involved.

You can also improve the utility of CGIWRAP by modifying the sample. Add your own Command Line options, including, perhaps, a "record" operation when no Command Line options are specified. A record operation automates the process of setting environment variables and piping standard input. The idea is to run CGIWRAP as a normal CGI application from your calling web page. CGIWRAP identifies that no Command Line parameters were sent, so it records the environment and saves it to a file. Then, from a debugger (on any machine), specify a Command Line option to CGIWRAP telling it to load its environment from the previously saved file. CGIWRAP doesn't do this now, but these changes are easy to make yourself.

Keep in mind that CGIWRAP is not exactly like IIS. For example, CGIWRAP does not run in the context of a local system. This can make a tremendous difference in the way your ISAPI DLL behaves. Fortunately, when your DLL works with CGIWRAP but not with IIS, you'll know the problem is very likely related to security.

Using ISmoke to Test an Extension

Another sample that is helpful in the development process is ISmoke. Use the ISmoke sample to stress-test your ISAPI DLL. It is quite good at detecting problems that arise when your DLL is used in production. While the latest version of ISmoke still only tests extensions, not filters,

it runs in the proper security and is multi-threaded.

Because ISmoke does not support ISAPI filters, bugs that are caused by filters won't show up. If your DLL works with ISmoke but has problems in IIS, verify that your development setup does not have a buggy ISAPI filter installed. If possible, test your ISAPI extension on an IIS setup that uses no filters.

Additional reference words: 3.51 4.00 iis isapi cgiwrap ismoke debug www cgi activex cache
KBCategory: kbprg kbtshoot
KBSubcategory: kbnocat

Tips for Installing TAPI Service Providers

PSS ID Number: Q131356

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0
- Microsoft Windows Telephony Software Development Kit (TAPI SDK) version 1.0

SUMMARY

When installing TAPI Service Providers (TSPs), there is more involved than just copying the TSP files to the system directory. The TAPI system files may need to be installed, and the TSP must be installed into TAPI. However, this process can vary greatly from one Windows version to another.

MORE INFORMATION

First and foremost, be sure to do version checking when installing all files.

Second, use a setup application rather than relying on an .INF file. While the ATSP sample uses a .INF file to install, there are two major problems with using a .INF file:

- TAPI needs to already be installed (which isn't guarenteed under Windows version 3.1 or Windows for Workgroups).

- You can't do operating system version checking if you use an .INF file.

Files Necessary for Windows Version 3.1 or Windows for Workgroups

When a TSP is installed under Windows version 3.1 or Windows for Workgroups, it must check and make sure all the necessary system files are present. The files to be installed can be found in the TAPI 1.0 SDK \REDIST directory and must be distributed with the TSP. Some of these are optional and some are not. Here is a list of files and where they should be placed:

Mandidory TAPI files:

```
    TELEPHON.CPL [system]
    TAPI.DLL     [system]
    TAPIADDR.DLL [system]
    TAPIEXE.EXE  [system]
    TELEPHON.HLP [system]
```

Optional TAPI files:

```
    ATSPEXE.EXE  [system]
    ATSP.TSP     [system]
```

```
    ATSP.HLP     [system]
    DIALER.EXE   [windows]
    DIALER.HLP   [windows]
```

Files Necessary for Windows 95

Because the Windows 95 system files are not redistributable and because TAPI is installed automatically under Windows 95, the TAPI system files must not be distributed with the TSP. The TAPI version 1.0 files are not compatible with Windows 95, so it is important that no TAPI system files are installed under Windows 95.

One complication for Windows 95 is the TELEPHON.CPL file. This file is installed to the system directory by default along with all the rest of the Windows 95 TAPI files. However, because this file is not needed by most people using Windows 95, it is installed in the system directory as TELEPHON.CP$ to reduce control panel clutter. TSPs that need this applet should first look in the system directory for TELEPHON.CPL; if that isn't found, the TSP should locate TELEPHON.CP$ (also looking in the system directory), and rename it to TELEPHON.CPL.

Installing the TSP

Once all the files are installed, the TSP must now be installed into TAPI. Under Windows version 3.1, this is done through the Telephony Control Panel applet (TELEPHON.CPL). There are no APIs to automate the installation, so TSP installation programs must either tell the user how to install their TSP into TAPI or use an application such as MSTEST that can simulate keystrokes to automate this part of the installation. Note that TSPs must have version information (as demonstrated by the ATSP sample) to show in the Add Driver dialog.

Adding TSPs through the Telephony Control Panel is also available under Windows 95, but the exact sequence of keystrokes is slightly different. However, several TAPI APIs (lineAddProvider, lineConfigProvider, and lineRemoveProvider) have been added that make the TELEPHON.CPL unnecessary. It's a good idea to have each TSP have its own control panel applet that calls these APIs, rather than rely on TELEPHON.CPL. The setup program should install the TSP directly by using these APIs instead of asking the user to run a control panel applet.

Additional reference words: 1.00 1.30 1.40 4.00 95
KBCategory: kbprg
KBSubcategory: TAPI

Tips for Writing Multiple-Language Scripts

PSS ID Number: Q89865

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY
=======

To aid you in writing multiple-language resources, the Win32 development system supports language scripts.

MORE INFORMATION
================

To create a multiple language script, first create a single-language script file (American English, for example), and duplicate the translations in your script file. You need a complete translation only once for each major language. Only those resources that have differences between the major language and the sublanguage need be included in the sublanguage areas of the script. The system will use the main language resource if it doesn't find a resource for the sublanguage.

Sample Code

```
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US
<original script file>
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_UK
<portions of script file that are different for UK>
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_AUS
<portions of script file that are different for Australia>
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_CAN
<portions of script file that are different for Canada>
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH
<entire script file translated to French>
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH_CAN
<portions of script file that are different for Canada>
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH_SWISS
<portions of script file that are different for Switzerland>
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH_BELGIAN
<portions of script file that are different for Belgium>
LANGUAGE LANG_SPANISH,SUBLANG_SPANISH
<entire script file translated to Spanish>
```

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: WIntlDev

Tips for Writing Windows Sockets Apps That Use AF_NETBIOS

PSS ID Number: Q129316

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.5

SUMMARY

The six issues listed in this article must be addressed when writing Windows Sockets applications that use the AF_NETBIOS protocol family.

NOTE: Windows 95 does not support AF_NETBIOS.

MORE INFORMATION

Issues to Address When Writing Windows Sockets Applications

1. When creating an AF_NETBIOS socket, specify -1 times lana for the protocol option.

   The Windows NT WinSock library allows the programmer to create a socket that allows communication over a particular lana. The lana number is specified via the protocol option of the socket() function. To create an AF_NETBIOS socket, specify -1 times the lana for the protocol option of the socket() function. For example:

   ```
   SOCKET hSock0 = socket( AF_NETBIOS, SOCK_DGRAM, 0 );  // lana 0;
   SOCKET hSock1 = socket( AF_NETBIOS, SOCK_DGRAM, -1 ); // lana 1;
   ```

   The lana numbers are basically indices to the list of transports supported by the NetBIOS implementation. A given host has one unique lana number for every installed transport that supports NetBIOS. For example, listed below are some possible lana numbers for a typical Windows NT configuration:

Lana	Transport
0	TCP/IP // lana 0 is the default NetBIOS transport
1	IPX/SPX w/NetBIOS
3	NetBEUI

   In the case of a multihomed host (a machine with multiple network adapters), the number of unique lana numbers equals the number of network transports that support NetBIOS times the number of network adapters. For example, if the machine depicted above contained two network adapters, it would have a total of 3 * 2 = 6 lana numbers.

   Also, please note the WSNETBS.H header included with the Windows NT

version 3.5 SDK erroneously defines the NBPROTO_NETBEUI symbol. This symbol cannot be used as a protocol option and should be ignored.

2. Use the snb_type values provided by the WSNETBS.H header for NetBIOS name registration and deregistration.

   When filling out a sockaddr_nb structure, you must specify the appropriate value for the snb_type field. This field is used during the bind() operation to handle NetBIOS name registration. The WSNETBS.H header defines several values for this field; however only the following two values are currently implemented:

   - NETBIOS_UNIQUE_NAME

     Registers a unique NetBIOS name. This action is usually performed by a client or a server to register an endpoint.

   - NETBIOS_GROUP_NAME

     Registers a group name. This action is typically performed in preparation for sending or receiving NetBIOS multicasts.

   Names are registered during the bind() operation.

3. Use the supported socket types of SOCK_DGRAM and SOCK_SEQPACKET.

   Due to the nature of NetBIOS connection services, SOCK_STREAM is not supported.

4. Choose a NetBIOS port that does not conflict with your network client software.

   The NetBIOS port is an eight-bit value stored in the last position of the snb_name that is used by various network services to differentiate various type of NetBIOS names. When you register NetBIOS names, choose port values that do not cause conflicts with existing network services. This is of particular importance if you are registering a NetBIOS name that duplicates a user name or a machine name on the network. The following lists the reserved port values:

   0x00, 0x03, 0x06, 0x1f, 0x20, 0x21, 0xbe, 0xbf, 0x1b, 0x1c, 0x1d, 0x1e

5. Applications should use all available lana numbers when initiating communication.

   Because the NetBIOS interface can take advantage of multiple transport protocols, it is important to use all lanas when initiating communication. Server applications should accept connections on sockets for each lana number, and client applications should attempt to connect on every available lana. In a similar fashion, data gram broadcasts should be sent from sockets created from each lana.

   The following diagram depicts the lana mappings for two machines. In order for a client application running on Machine A to communicate

with a server application on Host B, the client application must create a socket on lana 3, and the server must create a socket on lana 1. Because the client and the server cannot know in advance which single lana to use, they must create sockets for all lanas.

```
   Host A (Client)            Host B (Server)
   ---------------            ---------------
   lana  Transport            lana  Transport
   ----  ---------            ----  ---------

   0     NetBEUI              0     TCP/IP
   3     IPX/SPX <==========> 1     IPX/SPX
                              3     NetBEUI
```

The above diagram illustrates several other important points about lanas. First, a transport that has a certain lana number on one host does not necessarily have the same lana number on other machines. Second, lana numbers do not have to be sequential.

The EnumProtocols() function can be used to enumerate valid lana numbers. Listed below is a code fragment that demonstrates this type of functionality:

```
#include <nspapi.h>

      DWORD          cb = 0;
      PROTOCOL_INFO *pPI;
      BOOL           pfLanas[100];

      int            iRes,
                     nLanas = sizeof(pfLanas) / sizeof(BOOL);

      // Specify NULL for lpiProtocols to enumerate all protocols.

      // First, determine the output buffer size.
      iRes = EnumProtocols( NULL, NULL, &cb );

      // Verify the expected error was received.
      // The following code must appear on one line.
      assert( iRes == -1 && GetLastError() ==
                  ERROR_INSUFFICIENT_BUFFER );

      if (!cb)
      {
         fprintf( stderr, "No available NetBIOS transports.\n");
         break;
      }

      // Allocate a buffer of the specified size.
      pPI = (PROTOCOL_INFO*) malloc( cb );

      // Enumerate all protocols.
      iRes = EnumProtocols( NULL, pPI, &cb );

      // EnumProtocols() lists each lana number twice, once for
```

```
         // SOCK_DGRAM and once for SOCK_SEQPACKET. Set a flag in pfLanas
         // so unique lanas can be identified.

         memset( pfLanas, 0, sizeof( pfLanas ));

         while (iRes > 0)
            // Scan protocols looking for AF_NETBIOS
            if ( pPI[--iRes].iAddressFamily == AF_NETBIOS )
               // found one.
               pfLanas[ pPI[iRes].iProtocol ] = TRUE;

         fprintf( stderr, "Available NetBIOS lana numbers: " );
         while( nLanas-- )
            if ( pfLanas[nLanas] )
               fprintf( stderr, "%d ", nLanas );

         free( pPI );
      }
```

6. Performance: Use of AF_NETBIOS is recommended for communication with down-level clients.

   On Windows NT, NetBIOS is a high level emulated interface. Consequently, applications that use the WinSock() function over NetBIOS obtain lower throughput than applications that use WinSock() over a native transport such as IPX/SPX or TCP/IP. However, due to the simplicity of the WinSock interface, it is a desirable interface for writing new 32-bit applications that communicate with NetBIOS applications running on down-level clients like Windows for Workgroups or Novell Netware.

Additional reference words: 3.50
KBCategory: kbnetwork kbcode
KBSubcategory: NtwkWinsock

Top-Level Menu Items in Owner-Draw Menus

PSS ID Number: Q69969

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Microsoft Windows version 3.0 allows an application to specify owner-draw menus. This provides the application with complete control over the appearance of items in the menu. However, Windows 3.0 only supports owner-draw items in a pop-up menu. Top-level menu items with the MF_OWNERDRAW style do not work properly.

In Windows 3.1, Windows NT, and Windows 95, top-level menu items with the MF_OWNERDRAW style work properly.

MORE INFORMATION

An application may append an item with the MF_OWNERDRAW style to a top-level menu. At this point, the parent application should receive a WM_MEASUREITEM message and a WM_DRAWITEM message.

However, a WM_MEASUREITEM message is never sent to the parent window for the menu item. In addition, the WM_DRAWITEM message is sent only when the selection state of the item changes (the action field in the DRAWITEMSTRUCT is equal to ODA_SELECTED). The WM_DRAWITEM message is not sent with the action field in the DRAWITEMSTRUCT equal to ODA_DRAWENTIRE.

Additional reference words: 3.00 3.10 3.50 4.00
KBCategory: kbui
KBSubcategory: UsrMen

Tracking Brush Origins in a Win32-based Application

PSS ID Number: Q102353

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

When writing to the Win32 API on Windows NT, it is no longer necessary to keep track of brush origins yourself. GDI32 keeps track of the brush origins by automatically recognizing when the origin has been changed. In Windows, you have to explicitly tell GDI to recognize the change by calling UnrealizeObject() with a handle to the brush. When a handle to a brush is passed to UnrealizeObject() in Windows NT, the function does nothing, therefore, it is still safe to call this API.

Win32s and Windows 95 require that you track the brush origins yourself, just as Windows 3.x does.

In Windows 3.1, the default brush origin is the screen origin. In Windows NT, the default origin is the client origin.

Additional reference words: 3.10 3.50 4.00
KBCategory: kbgraphic
KBSubcategory: GdiPnbr

Translating Client Coordinates to Screen Coordinates

PSS ID Number: Q11570

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The GetClientRect function always returns the coordinates (0, 0) for the origin of a window. This behavior is documented in the "Microsoft Windows Software Development Kit (SDK) Programmer's Reference" manual.

MORE INFORMATION

To determine the screen coordinates for the client area of a window, call the ClientToScreen function to translate the client coordinates returned by GetClientRect into screen coordinates. The following code demonstrates how to use the two functions together:

```
RECT rMyRect;

GetClientRect(hwnd, (LPRECT)&rMyRect);
ClientToScreen(hwnd, (LPPOINT)&rMyRect.left);
ClientToScreen(hwnd, (LPPOINT)&rMyRect.right);
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Translating Dialog-Box Size Units to Screen Units

PSS ID Number: Q74280

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

In the Microsoft Windows graphical environment, the MapDialogRect function converts dialog-box units to screen units easily.

MORE INFORMATION
================

When a application dynamically adds a child window to a dialog box, it may be necessary to align the new control with other controls that were defined in the dialog box's resource template in the RC file. Because the dialog box template defines the size and position of a controls in dialog-box units rather than in screen units (pixels), the application must translate dialog-box units to screen units to align the new child window.

An application can use the following two methods to translate dialog-box units to screen units:

1. The MapDialogRect function provides the easier method. This function converts dialog-box units to screen units automatically.

   For more details on this method, please see the documentation for the MapDialogRect function in the Microsoft Windows Software Development Kit (SDK).

2. Use the GetDialogBaseUnits function to retrieve the size of the dialog base units in pixels. A dialog unit in the x direction is one-fourth of the width that GetDialogBaseUnits returns. A dialog unit in the y direction is one-eighth of the height that the function returns.

   For more details on this method, see the documentation for the GetDialogBaseUnits function in the Windows SDK.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 convert unit pixel
KBCategory: kbui
KBSubcategory: UsrDlgs

Transparent Blts in Windows NT

PSS ID Number: Q89375

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY

=======

In order to perform a transparent blt in Microsoft Windows versions 3.0 and 3.1, the BitBlt() function must be called two or more times. This process involves nine steps. (For more information on this process, see article Q79212 in the Microsoft Knowledge Base.)

Windows NT introduces a new method of achieving transparent blts. This method involves the use of the MaskBlt() function. The MaskBlt() function lets you use any two arbitrary ROP3 codes (say, SRCCOPY and BLACKNESS) and apply them on a pel-by-pel basis using a mask.

MORE INFORMATION

================

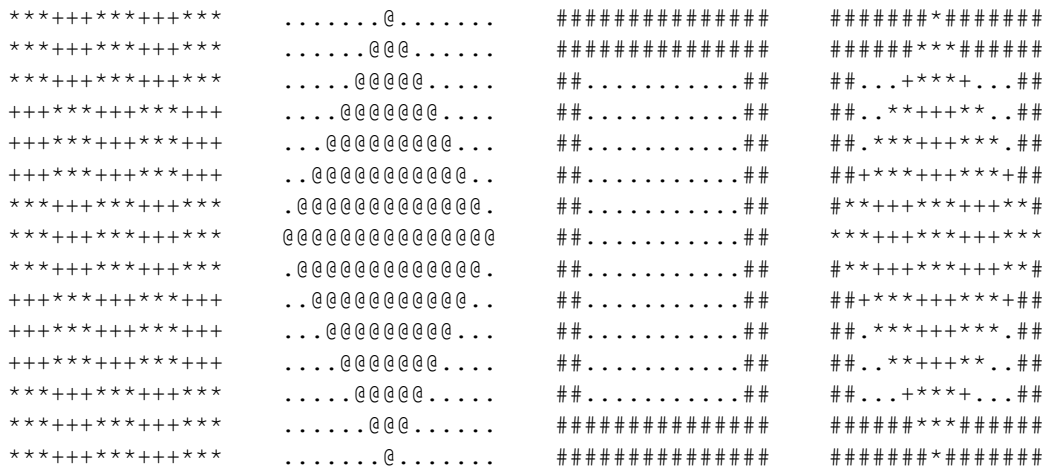
For this example, the source and target bitmaps contain 4 BPP. The call to the MaskBlt() function is as follows:

```
   MaskBlt(hdcTrg,  // handle of target DC
                0,  // x coord, upper-left corner of target rectangle
                0,  // y coord, upper-left corner of target rectangle
               15,  // width of source and target rectangles
               15,  // height of source and target rectangles
           hdcSrc,  // handle of source DC
                0,  // x coord, upper-left corner of source rectangle
                0,  // y coord, upper-left corner of source rectangle
          hbmMask,  // handle of monochrome bit-mask
                0,  // x coord, upper-left corner of mask rectangle
                0,  // y coord, upper-left corner of mask rectangle
        0xAACC0020  // raster-operation (ROP) code
                );
```

The legend is as follows

```
   '.' = 0,
   '@' = 1,
   '+' = 2,
   '*' = 3,
   '#' = 15
```

Source Bitmap	Mask Bitmap	Target Bitmap	Result



Note that the ROP "AA" is applied where 0 bits are in the mask and the ROP "CC" is applied where 1 bit is in the mask. This a transparency.

When creating a ROP4, you can use the following macro:

```
#define ROP4(fore,back) ((((back) << 8) & 0xFF000000) | (fore))
```

This macro can be used to call the MaskBlt() function as follows:

```
MaskBlt(hdcDest, xTrgt, yTrgt,
                        cx, cy,
           hdcSrc, xSrc, ySrc,
         hbmMask, xMask, yMask,
       ROP4(PATCOPY, NOTSRCCOPY)
                            );
```

This call would draw the selected brush where 1 bit appears in the mask and bitwise negation of the source bitmap where 0 bits appear in the mask.

Additional reference words: 3.10 3.50 pixel
KBCategory: kbgraphic
KBSubcategory: GdiBmp

Transparent Windows

PSS ID Number: Q92526

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Microsoft Windows version 3.1 does not support fully functional transparent windows.

MORE INFORMATION
================

If a window is created using CreateWindowEx() with the WS_EX_TRANSPARENT style, windows below it at the position where the original window was initially placed are not obscured and show through. Moving the WS_EX_TRANSPARENT window, however, results in the old window background moving to the new position, because Windows does not support fully functional transparent windows.

WS_EX_TRANSPARENT was designed to be used in very modal situations and the lifetime of a window with this style must be very short. A good use of this style is for drawing tracking points on the top of another window. For example, a dialog editor would use it to draw tracking points around the control that is being selected or moved.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Trapping Floating-Point Exceptions in a Win32-based App

PSS ID Number: Q94998

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

The _controlfp() function is the portable equivalent to the _control87() function.

To trap floating-point (FP) exceptions via try-except (such as EXCEPTION_FLT_OVERFLOW), insert the following before doing FP operations:

```
   // Get the default control word.
   int cw = _controlfp( 0,0 );

   // Set the exception masks OFF, turn exceptions on.
   cw &=~(EM_OVERFLOW|EM_UNDERFLOW|EM_INEXACT|EM_ZERODIVIDE|EM_DENORMAL);

   // Set the control word.
   _controlfp( cw, MCW_EM );
```

This turns on all possible FP exceptions. To trap only particular exceptions, choose only the flags that pertain to the exceptions desired.

Note that any handler for FP errors should have _clearfp() as its first FP instruction.

MORE INFORMATION

================

By default, Windows NT has all the FP exceptions turned off, and thus computations result in NAN or INFINITY rather than an exception. Note, however, that if an exception occurs and an explicit handler does not exist for it, the default exception handler will terminate the process.

If you want to determine which mask bits are set and which are not during exception handling, you need to use _clearfp() to clear the floating-point exception. This routine returns the existing FP status word, giving the necessary information about the exception. After this, it is safe to query the chip for the state of its control word with _controlfp(). However, as long as an unmasked FP exception is active, most FP instructions will fault, including the fstcw in _controlfp().

Additional reference words: 3.10 3.50 4.00 95

KBCategory: kbprg

KBSubcategory: BseExcept

Treeviews Share Image Lists by Default

PSS ID Number: Q131287

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY
=======

The LVS_SHAREIMAGELISTS style is available for listview controls to enable the same image lists to be used with multiple listview controls. There is no equivalent TVS_SHAREIMAGELISTS, however, for the treeview control because treeviews share image lists by default.

MORE INFORMATION
================

By default, a listview control takes ownership of the image lists associated with it. For listviews, this could be any of three image lists (those with large icons, small icons, or state images). These three image lists are set by calling ListView_SetImageList() with the flags LVSIL_NORMAL, LVSIL_SMALL, or LVSIL_STATE respectively. The listview then takes responsibility for destroying these image lists when it is destroyed.

Setting the LVS_SHAREIMAGELISTS style, however, moves ownership of the image lists from the listview control to the application. Because this style assumes that the image lists are shared by multiple listviews, specifying this style requires that the application destroy the image lists when the last listview using them is destroyed. Failure to do so causes a memory leak in the system.

Similarly, because treeviews share image lists by default, an application that associates an image list with a treeview should ensure that the image list is destroyed after the last treeview using it is destroyed. Another way to do this is to first set the treeview's image list to NULL by using TreeView_SetImageList(); then destroy the handle to the previous image list returned by the function.

Additional reference words: 1.30 4.00 95 Common Controls
KBCategory: kbui
KBSubcategory: UsrCtl

TrueType Font Converters and Editors

PSS ID Number: Q87817

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The text below lists a number of commercial software tools that convert existing fonts to TrueType fonts or help build new fonts. The tools are listed in alphabetical order by name. None of these tools is recommended over any other, nor over any that are absent from the list. Each of the tools has its own strengths and weaknesses. Before making any purchase, examine the tool and its documentation to see if it meets your needs. This list will be updated as more tools become available.

Some of the following tools run on an Apple Macintosh while others run on an IBM PC/AT or compatible computer. The same TrueType font can run on a Macintosh or with Microsoft Windows operating system version 3.1.

To convert a font from the Macintosh to Windows 3.1, save the data fork from a Macintosh font to a file, copy the file to an MS-DOS-formatted disk, give the file a .TTF file extension, and use the Windows 3.1 Control Panel to install the font.

The products included here are manufactured by vendors independent of Microsoft; we make no warranty, implied or otherwise, regarding these products' performance or reliability.

MORE INFORMATION
================

Last Update: 24 July 1992

FONT CONVERSION UTILITIES

AllType for MS-DOS

Author: Atech Software
5964 La Place Court, Suite 125
Carlsbad, CA 92008
(619) 438-6883

Description: Character-based application running under MS-DOS. Converts almost any font format to any other. Supported formats include TrueType, Type-1, Type-3, Nimbus-Q, and Intellifont.

Evolution 2.0 for Macintosh

Author: Image Club Graphics, Inc.
1902 11th St. SE, Suite 5
Calgary, Alberta, Canada T2G 3G2
(403) 262-8008
Description: Converts almost any font format to any other. Supported formats include TrueType, Type-1, and Type-3.

FontMonger for Windows
FontMonger for Macintosh

Author: Ares Software
561 Pilgrim Drive, Suite D
Foster City, CA 94404
(415) 578-9090
Description: Converts almost any font format to any other. Supported formats include TrueType, Type-1, Type-3, and Intellifont. Also provides minor font editing by creating composite characters or rearranging the characters in a font.

Incubator for Windows

Author: Type Solutions, Inc.
91 Plaistow Rd
Plaistow, NH 03865
(603) 382-6400
Description: Supports adding effects to TrueType fonts. Contact Type Solutions for more information.

Metamorphosis Professional for Macintosh

Author: Altsys Corp.
269 W. Renner Rd
Richardson, TX 75080
(214) 680-2060
Description: Converts almost any font format to any other. Supported formats include TrueType, Type-1, Type-3, and PICT format. One interesting feature allows the user to read a Type-1 font from the ROM of an Apple LaserWriter printer and convert the font to another format.

FONT EDITORS

Fontographer 3.5 for Windows
Fontographer 4.1 for Macintosh

Author: Altsys Corp.
269 W. Renner Rd
Richardson, TX 75080
(214) 680-2060
Description: A complete font editing tool. Supports creating a font from scratch and modifying existing fonts. Supports a variety of formats including TrueType and Type-1. Includes an autohinter.

FontStudio 2.0 for Macintosh

Author: Letraset Graphic Design Software
40 Eisenhower Dr
Paramus, NJ 07653
(800) 343-TYPE or (201) 845-6100
Description: A complete font editing tool. Supports creating fonts from scratch and modifying existing fonts. Supports a variety of formats including TrueType and Type-1. Includes an autohinter.

TypeMan 1.0 for Macintosh

Author: Type Solutions, Inc.
91 Plaistow Rd
Plaistow, NH 03865
(603) 382-6400
Description: Designed for font foundries, this tool provides precise control over the hints in a font and includes an autohinter. Supports specifying a font in a high-level programming language, which the tool compiles to the binary TrueType format.

Additional reference words: 3.10 3.50 4.00 95 true type
KBCategory: kbgraphic
KBSubcategory: GdiTt

Trusted DDE Shares

PSS ID Number: Q128125

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

SUMMARY

=======

To allow someone else to connect to DDE shares when you are logged in, you have to trust your existing DDE share. The reason is that when the other person connects to the share remotely, the application he will connect to is running in your security context, not the remote user's, because you are the logged-on user. You need to give permission for the other person to access the share. Even another person who is an administrator cannot trust a share for your account.

In your code, you would use NDdeShareAdd() to create the share and NDdeSetTrustedShare() to trust the share.

Alternatively, you can use DDESHARE to create the share. For more information on DDESHARE, please see the following article in the Microsoft Knowledge Base:

   ARTICLE-ID: Q114089
   TITLE   : Using the Windows NT NetDDE Share Manager

If you have a DDE share that always needs to be available, you can write a program that trusts the specific shares and sets up a logon script to trust the share for every user. The logon script should be a .BAT file that calls the .EXE file, so that you can add other things to the logon script as necessary.

MORE INFORMATION

================

The online documentation for NDdeSetTrustedShare says:

   The NDdeSetTrustedShare function is called to promote the
   referenced DDE share to trusted status within the current
   user's context.

DDE shares are a machine resource, not an account resource, just as shared drives are. However, NetDDE runs an application that must run in the context of the current user. This is the reason that the share must be trusted, so that the application can run in the user's context.

The prototype for the function is:

   UINT NDdeSetTrustedShare(lpszServer, lpszShareName, dwTrustOptions)

The parameter lpszServer is the address of the server name on which the DDE share resides. The DDE Share Database (DSDM) will be modified. This service manages the shared DDE conversations and is used by the NetDDE service. This parameter will generally be the current machine, because you can't trust a share for someone else.

The dwTrustOptions are NDDE_TRUST_SHARE_START and NDDE_TRUST_SHARE_INIT. NDDE_TRUST_SHARE_START allows the DDE server, such as Excel, to be started in the user's context. This allows a DDE client to make a NetDDE connection without the DDE server already running. When the NetDDE agent on the server machine detects the attempted connection, it launches the associated DDE server application if it is not already running.

NDDE_TRUST_SHARE_INIT allows a client to initiate to the DDE server if it is already executing in the user's context. This allows a DDE client to make a NetDDE connection to a DDE server already running on the server machine. If the DDE server is not already running, the connection will fail.

Additional reference words: 3.50
KBCategory: kbui
KBSubcategory: UsrNetDde

Types of File I/O Under Win32

PSS ID Number: Q99173

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

There are multiple types of file handles that can be opened using the Win32 API and the C Run-time:

Returned Type	File Creation API	API Set
HANDLE	CreateFile()	Win32
HFILE	OpenFile()/_lcreat()	Win32
int	_creat()/_open()	C Run-time
FILE *	fopen()	C Run-time

In general, these file I/O "families" are incompatible with each other. On some implementations of the Win32 application programming interfaces (APIs), the OpenFile()/_lcreat() family of file I/O APIs are implemented as "wrappers" around the CreateFile() family of file I/O APIs, meaning that OpenFile(), _lcreat(), and _lopen() end up calling CreateFile(), returning the handle returned by CreateFile(), and do not maintain any state information about the file themselves. However, this is an implementation detail only and is NOT a design feature.

NOTE: You cannot count on this being true on other implementations of the Win32 APIs. Win32 file I/O APIs may be written using different methods on other platforms, so reliance on this implementation detail may cause your application to fail.

The rule to follow is to use one family of file I/O APIs and stick with them--do not open a file with _lopen() and read from it with ReadFile(), for example. This kind of incorrect use of the file I/O APIs can easily be caught by the compiler, because the file types (HFILE and HANDLE respectively) are incompatible with each other and the compiler will warn you (at warning level /w3 or higher) when you have incorrectly passed one type of file handle to a file I/O API that is expecting another, such as passing an HFILE type to ReadFile(HANDLE, ...) in the above example.

MORE INFORMATION

Compatibility

The OpenFile() family of file I/O functions is provided only for

compatibility with earlier versions of Windows. New Win32-based applications should use the CreateFile() family of file I/O APIs, which provide added functionality that the earlier file I/O APIs do not provide.

Each of the two families of C Run-time file I/O APIs are incompatible with any of the other file I/O families. It is incorrect to open a file handle with one of the C Run-time file I/O APIs and operate on that file handle with any other family of file I/O APIs, nor can a C Run-time file I/O family operate on file handles opened by any other file I/O family.

_get_osfhandle()

For the C Run-time unbuffered I/O family of APIs [_open(), and so forth], it is possible to extract the operating system handle that is associated with that C run-time handle via the _get_osfhandle() C Run-time API. The operating system handle is the handle stored in a C Run-time internal structure associated with that C Run-time file handle. This operating system handle is the handle that is returned from an operating system call made by the C Run-time to open a file [CreateFile() in this case] when you call one of the C Run-time unbuffered I/O APIs [_open(), _creat(), _sopen(), and so forth].

The _get_osfhandle() C Run-time call is provided for informational purposes only. Problems may occur if you read or write to the file using the operating system handle returned from _get_osfhandle(); for these reasons we recommend that you do not use the returned handle to read or write to the file.

_open_osfhandle()

It is also possible to construct a C Run-time unbuffered file I/O handle from an operating system handle [a CreateFile() handle] with the _open_osfhandle() C Run-time API. In this case, the C Run-time uses the existing operating system handle that you pass in rather than opening the file itself. It is possible to use the original operating system handle to read or write to the file, but it is very important that you use only the original handle or the returned C Run-time handle to access the file, but not both, because the C Run-time maintains state information that will not be updated if you use the operating system handle to read or write to the file.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseFileio

Types of Thunking Available in Win32 Platforms

PSS ID Number: Q125710

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.50 and 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SUMMARY

=======

Thunks allow code on one side of the 16-32 process boundary to call into code on the other side of the boundary. Each Win32 platform employs one or more thunking mechanisms. This table summarizes the thunking mechanisms provided by the different Win32 platforms.

	Win32s	Windows 95	Windows NT
Generic Thunk		X	X
Universal Thunk	X		
Flat Thunk		X	

Generic Thunks allow a 16-bit Windows-based application to load and call a Win32-based DLL on Windows NT and Windows 95.

Windows 95 also supports a thunk compiler, so a Win32-based application can load and call a 16-bit DLL.

Win32s Universal Thunks allow a Win32-based application running under Win32s to load and call a 16-bit DLL. You can also use UT to allow a 16-bit Windows-based application to call a 32-bit DLL under Win32s, but this isn't officially supported. Certain things do not work on the 32-bit side because the app was loaded with the context of a 16-bit Windows-based application.

This article describes the types of thunking mechanisms available on each Win32 platform.

MORE INFORMATION

================

Windows NT

Windows NT supports Generic Thunks, which allow 16-bit code to call into 32-bit code. Generic Thunks must be initiated from a 16-bit Windows-based application. Once the thunk is established, the 32-bit code can make a callback to the 16-bit code using WOWCallback16(). The generic thunk is

implemented by using a set of API functions that are exported by the WOW KERNEL and WOW32.DLL.

In Windows NT, 16-bit Windows-based applications are executed in a subsystem (or environment) called WOW (Windows On Win32). Each application runs as a thread in a VDM (virtual DOS machine).

Using generic thunks is like explicitly loading a DLL. The four major APIs used in generic thunking are: LoadLibraryEx32W(), FreeLibrary32W(), GetProcAddress32W(), and CallProc32W(). Their functionality is very similar to LoadLibraryEx(), FreeLibrary(), GetProcAddress(), and calling the function through a function pointer. The Win32-based DLL called by the thunk is loaded into the VDM address space. The following is a example of thunking a call to GetVersionEx():

```
   void FAR PASCAL __export MyGetVersionEx(OSVERSIONINFO *lpVersionInfo)
   {
      HINSTANCE32 hKernel32;
      FARPROC lpGetVersionEx;

      // Load KERNEL32.DLL
      if (!(hKernel32 = LoadLibraryEx32W("KERNEL32.DLL", NULL, NULL)))
      {
         MessageBox(NULL, "LoadLibraryEx32W Failed", "DLL16", MB_OK);
         return;
      }

      // Get the address of GetVersionExA in KERNEL32.DLL
      if (!(lpGetVersionEx =
            GetProcAddress32W( hKernel32, "GetVersionExA")))
      {
         MessageBox(NULL, "GetProcAddress32W Failed", "DLL16", MB_OK);
         return;
      }
      lpVersionInfo->dwOSVersionInfoSize = sizeof(OSVERSIONINFO);

      // Call GetVersionExA
      CallProc32W(lpVersionInfo, lpGetVersionEx, 1, 1);

      // Free KERNEL32.DLL
      if (!FreeLibrary32W(hKernel32))
      {
         MessageBox(NULL, "FreeLibrary32W Failed", "DLL16", MB_OK);
         return;
      }
      return;
   }
```

Win32s

All 16-bit Windows-based and Win32-based applications run in a single address space in Win32s. The mechanism that is provided for accessing 16-bit code from 32-bit code is called the Universal Thunk. The Universal Thunks consists of 4 APIs. The major APIs, UTRegister() and UTUnRegister(),

are exported by KERNEL32. The prototype for UTRegister() is:

```
   BOOL UTRegister(HANDLE hModule,        // Win32-based DLL handle
                   LPCTSTR lpsz16BITDLL,  // 16-bit DLL to call
                   LPCTSTR lpszInitName,  // thunk initialization procedure
                   LPCTSTR lpszProcName,    // thunk procedure
                   UT32PROC *ppfn32Thunk,   // pointer to thunk procedure
                   FARPROC pfnUT32CallBack, // optional callback
                   LPVOID lpBuff);          // shared memory buffer
```

NOTES: lpszInitName, pfnUT32CallBack, and lpBuff are optional parameters. The value for ppfn32Thunk is the returned value of the 32-bit function pointer to the thunk procedure. The buffer lpBuff is a globally allocated shared memory buffer that is available to the 16-bit initialization routine via a 16-bit selelctor:offset pointer.

The function pointer returned in ppfn32Thunk has the following syntax:

```
   WORD (*ppfn32Thunk)(lpBuff, dwUserDefined, *lpTranslationList);
```

where lpBuff is the pointer to the shared data area, dwUserDefined is available for application use (it is most commonly used as a switch for multiple thunked functions), and lpTranslationList is an array of flat pointers within lpBuff that are to be translated into selector:offset pointers.

This method is not portable to other platforms.

Windows 95

Thunking in Windows 95 allows 16-bit code to call 32-bit code and vice-versa. The mechanism used is a thunk compiler. To use the thunk compiler you need to create a thunk script, which is the function prototype with additional information about input and output variables. NOTE: These thunks are not portable to other platforms.

The thunk compiler produces a single assembly language file. This single assembly language file should be assembled using two different flags - DIS_32 and -DIS_16 to produce a 16-bit and 32-bit object files. These object modules should be liked to their respective 16-bit and 32-bit DLL's. There are no special APIs used, all you have to do is call the function.

In addition to Flat Thunks, Windows 95 supports the Windows NT Generic Thunk mechanism. Generic thunks are recommended for portability between Windows 95 and Windows NT.

REFERENCES
==========

For more information on Generic Thunks, see GENTHUNK.TXT on the Win32 SDK CD.

For more information on the Universal Thunk, see the "Win32s Programmer's Reference" and the UTSAMP sample on the Win32 SDK CD.

Additional reference words: 1.30 3.50 4.00
KBCategory: kbprg
KBSubcategory: SubSys BseMisc W32s

Understanding Disk Volume Tracking in Windows 95

PSS ID Number: Q150582

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  Microsoft Windows 95 version 4.0

SUMMARY
=======

When users insert a disk in a floppy disk drive, Windows 95 tracks the disk to prevent the user from opening a file, removing the disk, and inadvertently inserting a different disk. Through this behavior, Windows 95 helps users guard against corrupting the data on their disks. This article explains how Windows 95 tracks disks and discusses instances where software developers can consider overriding the default mechanism Windows 95 uses to perform this task.

MORE INFORMATION
================

Volume Tracking Overview

One part of the Windows 95 disk system is a driver called the Volume Tracking Driver. Its purpose is to uniquely identify disks and ensure that the correct disk is in the floppy disk drive when data is written to a file. The Volume Tracking Driver intercepts all writes to the floppy disk drive and, when it detects an invalid write operation, displays a text-mode blue screen prompting users to either insert the original disk or cancel the invalid write operation. There is no way to prevent Windows 95 from displaying this blue screen because it serves as a warning to users that an error has been made that could result in data loss.

The Volume Tracking Driver uniquely identifies disks by writing a volume tracking serial number in the OEM ID field (offsets 0x3-0xB) of the boot record, which is stored in the boot sector of the disk. This serial number is different than the volume serial number created when the disk is formatted and is used solely by the Volume Tracking Driver. The Volume Tracking Driver assigns a volume tracking serial number the first time a disk is inserted since Windows was last started. That number identifies that particular disk until Windows 95 is restarted.

How to Override the Default Volume Tracking Method

There are times when the Volume Tracking Driver cannot or should not be allowed to overwrite the OEM ID field of the boot record of a disk. For example, write-protected disks physically prevent the Volume Tracking Driver from overwriting the OEM ID field. Also, some software programs, such as backups, rely on the OEM ID field to determine whether the disk contains valid data. For instance, if a utility program stores a name in

the OEM ID field and the Volume Tracking Driver overwrites this information, the disk becomes useless if the backup software checks for the name.

When the Volume Tracking Drive is prevented from overwriting the OEM ID field on a disk, it stores the disk's volume serial number and label in memory and uses the combination of these to identify the disk. Although this method does not modify the disk, it is slower than storing a unique ID on the disk itself.

Although the Volume Tracking Driver cannot be disabled, it can be prevented from modifying the OEM ID field of specific types of disks if the following value is created for the registry key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\control\
   FileSystem\NoVolTrack
```

The values in this key form a table of patterns and offsets that the Volume Tracking Driver uses to determine which disks should not be modified. When a new disk is inserted, the Volume Tracking Driver scans this table looking for each pattern at its corresponding offset in the disk's boot sector. If it finds a pattern, it does not update the OEM ID field in the disk's boot record, but rather identifies the disk by its volume serial number and label.

Each value of the NoVolTrack key is binary data formatted as a two-byte offset in little-endian order followed by an arbitrarily long sequence of bytes that make up a pattern. The Volume Tracking Driver ignores the label of each value, but the labels are used to aid human readers in understanding the type of disk to which the value refers. An example table of values looks like the following:

```
NoVolTrack
   Label    Offset   Pattern
   -------  ------   --------------------
   MyDisk   0050h    MyDisk
   COOL     0100h    COOL
```

In the Registry Editor, this table looks like the following:

```
   MyDisk   50 00 4D 79 44 69 73 6B
   COOL     00 01 43 4F 4F 4C
```

If your software relies on the OEM ID in the boot record of disks it uses, you need to register your "disk type" by adding it to the table of values in the NoVolTrack registry key.

Additional reference words: 4.00 floppy disk boot sector diskette
KBCategory: kbprg kbhowto
KBSubcategory: BseFileio

Understanding Volume-Level Security on Windows NT

PSS ID Number: Q150101

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with
  Microsoft Windows NT, versions 3.5, 3.51, 4.0

SUMMARY
=======

This article describes how volume-level security behaves, and how it differs from the file- and directory-level security provided by the New Technology File System (NTFS).

Microsoft Windows NT provides auditing and access control facilities for physical and logical volumes. Conceptually, volume-level security is designed to control volume-level access, such as formatting operations, rather than file- and directory-level operations. Volume-level security also minimally protects logical volumes that are not formatted with NTFS, such as floppy disks and compact discs.

MORE INFORMATION
================

Windows NT was designed to have security descriptors applied to volumes control and audit volume-level access such as formatting a disk or reading raw disk sectors. The file system with which the logical volume is formatted is responsible for control and auditing of access to files and directories. Currently, the only file system to support file- and directory-based security is NTFS.

There are two types of volume-level operations that need to be understood: physical drive access and logical volume access. Physical drive access allows Win32-based applications to manage hard disk partition tables and logical volumes. Logical volume access allows Win32-based applications to manage the contents of a partition below the file system level. Logical volume access is most often used to format a partition with a particular file system, such as FAT or NTFS. The following sections explain how volume-level security applies to each type of volume-level access.

Physical Drive Security

Win32-based applications can open handles to physical hard disk drives for low-level access by calling CreateFile() with \\.\PHYSICALDRIVEx, where x is the zero-based physical drive number. Once the applications have a handle, they can use DeviceIoControl() to call IOCTL functions, or ReadFile() and WriteFile() to perform sector reads and writes.

Access to physical drives is a built-in right of the Administrators group and does not need to be enabled. Users who are not members of the Administrators group cannot open physical drives under any circumstances.

Although any member of the Administrators group can enable access control and auditing on physical drives, doing so provides little value for two reasons, the first leading to the second:

1. Adding access control lists (ACLs) to a physical drive does not control or audit access to the logical volumes on the drive, or to files and directories within those logical volumes. Instead, the ACLs are used to control and audit access to the physical drive itself, such as opening the physical drive with CreateFile() using \\.\PHYSICALDRIVEx.

2. Because only members of the Administrators group can access physical drives, the only use for an ACL on a physical drive is to control or audit the access that they have. Because members can change the ACLs on physical drives, their access is not truly limited.

Finally, the ACLs applied to physical drives are in effect only until the system is shut down or restarted.

Logical Volume Security

Win32-based applications can open logical volumes with CreateFile() by specifying the file name as \\.\X:, where X is the actual drive letter. Once this is done, the applications can use the handle to issue IOCTL functions or read and write raw sectors.

Specifying access control on a logical volume controls access to the logical volume itself and to all of the files and directories contained by the volume allows Windows NT to provide a basic level of security to media that is not formatted with NTFS, such as floppy disks, compact discs, and hard disk partitions formatted with the FAT file system.

Access controls applied to removable media affect access to the drive, not just to the media that was in the drive when the drive was secured. For example, if users are prevented from accessing A:\, they are not able to use drive A:, no matter which floppy disk is inserted.

Although access control protects a logical volume and all of its contents, auditing only records accesses to the volume itself. Accesses to files and directories within the volume are not recorded. This is analogous to the way NTFS audits directories; only access to the directory itself is audited. Thus, opening a logical volume by specifying \\.\X: as a file name to CreateFile(), causes an entry to be placed in the security event log while opening a file on the same volume does not.

Auditing and access control apply to logical volumes only as long as the system is running. If you want to control access to or audit logical volumes after the machine has been restarted, you must reapply the security.

How to Enable Volume-Level Security

Access control and auditing are enabled by placing access control lists

(ACLs) on the volume. The discretionary ACL (DACL) provides access control information, while the system ACL (SACL) provides auditing. To apply ACLs to a volume, create a security descriptor with the desired ACLs, and then use SetFileSecurity() to apply it. SetFileSecurity() can be used on both physical and logical volumes and in addition to files and directories, as follows:

- Physical volumes must be specified as a string in the form \\.\PHYSICALDRIVEx, where x is the zero-based drive number.

- Logical volumes must be specified as a string in the form \\.\x:, where x is the actual drive letter of the volume.

Additional reference words: 3.50 3.51 4.00 NTFS CD-ROM
KBCategory: kbprg
KBSubcategory: BseSecurity

Understanding Win16Mutex

PSS ID Number: Q125867

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SUMMARY

Windows 95 offers preemptive multithreaded scheduling for Win32 processes, yet also provides the familiar non-preemptive task switching found in Windows 3.x for Win16 applications. Some of the Win32 system DLLs, such as USER32.DLL and GDI32.DLL, thunk to their 16-bit counterparts for compatibility and size. Unlike Windows 3.x, which did not preemptively schedule processes, Windows 95 must be able to handle the possibility that it might be reentered by two or more processes each calling the same API functions. Windows 95 must do so in a way that uses little memory and is compatible with all existing 16-bit applications and DLLs.

Win16Mutex is a global semaphore that is used to protect the 16-bit Windows 95 components from being reentered by preventing Win32 threads from thunking to 16-bit components while other 16-bit code is running. Win16Mutex is internal to Windows 95 and is not accessible from applications or DLLs. This article explains how Win16Mutex works and offers design tips for minimizing the effects Win16Mutex may have on Win32 applications.

MORE INFORMATION

Because Windows 3.x is a non-preemptive system, Windows 3.x did not need to be designed to prevent the system from being reentered. Only one task (application instance) at a time can call system services (API functions) because other tasks cannot run until the active task voluntarily yields control of the CPU. Since only one task can execute at a time, it is not possible to have two different tasks calling the same API function, and thus Windows does not need to protect itself from reentrancy.

Windows 95 differs from Windows 3.x because Windows 95 provides support for both Win32 and Win16 applications. In Windows 95, every instance of every Win16 application is a process with exactly one thread, and every Win32 process has at least one thread. Win32 threads are preemptively scheduled and may even preempt Win16 processes. Because many Win32 API functions are thunked to 16-bit Windows API functions, there is now a possibility for the 16-bit Windows components to be reentered. Since the 16-bit Windows components are largely the same as in Windows 3.x, they need to be protected from being reentered.

The Win16Mutex provides this protection by allowing only one thread (not process) at a time to access the 16-bit APIs. Whenever Win16Mutex is owned

by a thread, any other thread that tries to claim Win16Mutex will block until Win16Mutex is released. Now the question remains: "When does Win16Mutex get claimed and released?"

Whenever a Win16 process is running, it owns Win16Mutex. That is, when a Win16 process first gets a message via GetMessage or PeekMessage, the Win16 process claims Win16Mutex. The Win16 process releases Win16Mutex whenever the process yields, such as when the process calls GetMessage or PeekMessage and doesn't return. The only way a Win16 process can keep Win16Mutex indefinitely is to never yield; since the message-processing mechanism provides the scheduler in 16-bit Windows, the only way to never yield is to stop processing messages (which makes the application unresponsive to user input).

The only time a thread in a Win32 process claims Win16Mutex is when the thread makes a call to an API function which thunks to one of the 16-bit Windows components or when the thread thunks directly to a Win16 DLL. Immediately after the call returns, the process releases Win16Mutex. Not all API functions thunk to 16-bit components; most 32-bit USER and GDI functions thunk to 16-bit USER and GDI, but none of the 32-bit KERNEL functions thunk to 16-bit KRNL386. One exception is when a Win32 process spawns a Win16 process, KERNEL32 thunks to KRNL386 to call the Win16 loader.

Putting 16-bit and 32-bit behaviors together, you can see that when a Win16 process is running, and a thread of a Win32 process preempts the Win16 process and calls a function which thunks to a 16-bit component, the Win32 thread is put to sleep until the Win16 process yields, which releases Win16Mutex. Likewise, when one Win32 process's thread claims Win16Mutex and then loses its timeslice, and another thread from either the same or a different process tries to claim Win16Mutex, the second thread blocks until Win16Mutex is released.

Win16Mutex is internal to Windows 95 and may not be manipulated or even checked by applications and DLLs. Win16Mutex is implemented mainly in the thunk layer so that every Win32 API which thunks to a Win16 component will automatically claim Win16Mutex before entering 16-bit code. Additionally, thunks created by the thunk compiler to allow Win32 applications and DLLs to call 16-bit DLLs claim Win16Mutex automatically, so programmers do not have to do so explicitly.

One way to lessen the impact that Win16 applications have on the responsiveness of Win32 applications is to create multiple threads where the primary thread (the initial thread of the process) controls the entire user interface for the process and each additional thread performs some useful task, such as reading or writing to a data file, but does not make user interface or graphics calls. This way, if the Win32 process's primary thread blocks waiting for a Win16 process to yield, its other threads are still performing useful work.

Additional reference words: 4.00
KBCategory: kbprg
KBSubcategory: BseProcThread

UNICODE and _UNICODE Needed to Compile for Unicode

PSS ID Number: Q99359

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

To compile code for Unicode, you need to #define UNICODE for the Win32 header files and #define _UNICODE for the C Run-time header files. These #defines must appear before the

```
#include <windows.h>
```

and any included C Run-time headers. The leading underscore indicates deviance from the ANSI C standard. Because the Windows header files are not part of this standard, it is allowable to use UNICODE without the leading underscore.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrNls

Unicode Conversion to Integers

PSS ID Number: Q89295

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY
=======

Under Windows NT, strings may be either Unicode or ANSI. There is no function for reliably converting a string that might be either Unicode or ANSI to an integer. This is because, given a random set of bytes, it is difficult to determine whether the string is in Unicode or ANSI. The calling program has to know which format the string uses in order to convert it.

If the string uses Unicode, the functions wcstol(), wcstoul(), and wcstod() can be used to perform the conversion.

Note that when you are using the Win32 application programming interface (API), you can choose what kind of characters you get from the console or window manager. The names of the API functions that are called to use Unicode and ANSI characters are different. For more details, see Chapter 93 in the overview, "Unicode."

To mark a string as Unicode, insert the byte-ordering-mark (BOM) 0xFEFF in the string and/or file.

MORE INFORMATION
================

You can assume that the first 128 bytes in each character set are in the same codepoint. For portability, you should code character conversions in this range as:

```
{
   TCHAR     c;
     ...
     i = c - TEXT('0');
}
```

The TEXT macro places an "L" before the constant if Unicode is defined.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: WIntlDev

Unicode Functions Supported by Windows 95

PSS ID Number: Q125671

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

Unlike Windows NT, Windows 95 does not implement the Unicode (or wide character) version of most Win32 functions that take string parameters. With some exceptions, these functions are implemented as stubs that simply return success without modifying any arguments.

In general, Windows 95 implements the ANSI version of these functions. See the Win32 API documentation for information on particular functions and differences between the various Win32 platforms.

One major exception to this rule is OLE. All native 32-bit OLE APIs and interface methods use Unicode exclusively. For more information on this, please see the OLE documentation.

Excluding OLE, Windows 95 supports the wide character version of the following functions:

   EnumResourceLanguages
   EnumResourceNames
   EnumResourceTypes
   ExtTextOut
   FindResource
   FindResourceEx
   GetCharWidth
   GetCommandLine
   GetTextExtentExPoint
   GetTextExtentPoint32
   GetTextExtentPoint
   lstrlen
   MessageBoxEx
   MessageBox
   TextOut

In addition, Windows 95 implements the following two functions for converting strings to or from Unicode:

   MultiByteToWideChar
   WideCharToMultiByte

Additional reference words: 4.00
KBCategory: kbprg
KBSubcategory: WIntlDev

Unicode Implementation in Windows NT 3.1 and 3.5

PSS ID Number: Q103977

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

Windows NT is the first widely available operating system to be built upon the Unicode character encoding. Almost all of the strings used in the system have 16-bits reserved for each character. However, Windows NT does not yet realize the Unicode ideal of offering an editor capable of handling one document containing all of the languages of the world.

MORE INFORMATION

Unicode support in Windows NT:

- All Windows USER objects support Unicode strings.
- The Win32 console is Unicode enabled.
- NTFS supports Unicode filenames.
- All of the information strings in the registry are Unicode.
- The L_10646.TTF (Lucida Sans Unicode) font covers over 1300 Unicode characters.
- Most of the TrueType fonts include a Unicode encoding table.

Unicode features missing from Windows NT:

- There is no font support for all of the Unicode characters.
- Although the Win32 console is Unicode enabled, it is not possible to use Unicode fonts in the console. Most Unicode characters will be represented by the "default character" of the System font.
- Winhlp32 is not Unicode enabled.
- There is no general Unicode input method in Windows NT version 3.1. The shell applets and File Manager fully support Unicode. You can use the new Notepad and Character Mapper applets to create files with Unicode text. (Choose the Lucida Sans Unicode font in the Character Mapper, then choose the desired Unicode characters in the Character Mapper and copy them to the clipboard. Paste the clipboard contents into Notepad, making sure Notepad has the Lucida Sans Unicode font selected, and save the file as a "Unicode Files". Note, this same process can be used to give files Unicode filenames.)
- The FAT and HPFS file systems do not support Unicode filenames. (Nor will they in the future; to accomplish this, use NTFS.)

Additional reference words: 3.10 3.50

KBCategory: kbui

KBSubcategory: UsrNls

Unimodem Only Supports INTERACTIVEVOICE for Outbound Calls

PSS ID Number: Q132192

The information in this article applies to:

- Microsoft Windows Telephony Software Development Kit (SDK) version 1.0, for Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK), versions 3.51 and 4.0

SUMMARY

Even though the Unimodem reports that it supports LINEMEDIAMODE_INTERACTIVEVOICE in LINEDEVCAPS.dwMediaMode, it only supports INTERACTIVEVOICE for outbound calls. If an application opens a line for both LINEMEDIAMODE_INTERACTIVEVOICE and LINECALLPRIVILEGE_OWNER, lineOpen fails with LINEERR_INVALMEDIAMODE.

MORE INFORMATION

Unimodem is designed to handle data calls, not voice calls. However, it does provide the ability to dial the phone for use with normal, interactive voice phone calls. This is done by supporting the LINEMEDIAMODE_INTERACTIVEVOICE media mode as indicated in LINEDEVCAPS.dwMediaMode. However, because Unimodem only supports dialing INTERACTIVEVOICE and not answering INTERACTIVEVOICE calls, it fails a lineOpen for LINEMEDIAMODE_INTERACTIVEVOICE that includes LINECALLPRIVILEGE_OWNER.

The suggested solution is to first attempt to lineOpen the line for LINEMEDIAMODE_INTERACTIVEVOICE and LINECALLPRIVILEGE_OWNER. If this fails, then attempt to open with LINECALLPRIVILEGE_NONE (or MONITOR privileges). If this succeeds, then the application can make, but not receive INTERACTIVEVOICE calls and should indicate this capability to the user.

Additional reference words: 1.00 4.00
KBCategory: kbprg
KBSubcategory: TAPI

Uniqueness Values in User and GDI Handles

PSS ID Number: Q94917

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

User and GDI handles, are divided into two parts:

- The lower 16 bits is an index into a system table of handle structures, which includes information such as the type of handle (window, menu, cursor, and so forth), as well as a value called the uniqueness.
- The upper 16 bits contain the same uniqueness value.

The first time a handle is issued by the system, the uniqueness value is 0 (zero). It is incremented each time the handle is re-used. In Windows NT 3.1, if you pass in a value of 0xFFFF for the uniqueness, the client side (that is, USER32.DLL) will look up the correct uniqueness value in shared memory and use the correct handle. In Windows NT 3.5, use 0x0000 for the uniqueness value.

This is important because it alleviates potential conflicts with re-used handles. For example, when a window is destroyed, its handle is reused by the system. The uniqueness value prevents an old handle to a destroyed window from being misinterpreted by the system as the handle to a new object, which was given the same handle value.

MORE INFORMATION

In Win32s, use 0x0000 in the upper 16 bits for the uniqueness.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrMisc

Use 16-Bit .FON Files for Cross-Platform Compatibility

PSS ID Number: Q100487

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

The AddFontResource() function installs a font resource in the GDI font table. Under Windows NT and Windows 95, the module can be a .FON file or a .FNT file. Under Windows 3.1, the module must be a .FON file. When using Win32s, AddFontResource() passes its argument to the Win16 AddFontResource, and therefore .FON files should be used for portability.

In addition, when running under Windows NT or Windows 95, the module can be either a 32-bit "portable executable" or a 16-bit .FON file. However, if the same Win32 executable is run under Win32s, the call to AddFontResource() fails if the .FON is not in 16-bit format. Therefore, for compatibility across platforms, use 16-bit .FON files. These can be created using the Windows 3.1 Software Development Kit (SDK).

Additional reference words: 3.10 3.50 4.00
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Use LoadLibrary() on .EXE Files Only for Resources

PSS ID Number: Q108448

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

The LoadLibrary() application programming interface (API) can be used to load an executable module. A common use of this function is to load a dynamic-link library (DLL), perform a subsequent call to GetProcAddress() to get the address of an exported DLL routine, and call this DLL routine through the address that is returned. Another use of LoadLibrary() is to load an executable module and retrieve its resources.

In Windows NT, a LoadLibrary() of an .EXE file is supported only for the purposes of retrieving resources. It was decided that it was rather uncommon to load an .EXE for any other purpose, so this limitation was imposed on LoadLibrary() to improve the performance in loading resources. Calling a routine in an .EXE through an address obtained with GetProcAddress() can cause an access violation.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Use MoveWindow to Move an Iconic MDI Child and Its Title

PSS ID Number: Q70079

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT version 3.5

Although Windows does not have application programming interface (API) calls to support dragging iconic windows (windows represented by an icon), calls to MoveWindow() can be used to implement a dragging functionality for iconic multiple-document interface (MDI) child windows.

However, the icon's title is not in the same window as the icon, so when the icon is moved, the title will not automatically move with it. The title is in a small window of its own, so a separate call to MoveWindow() must be made to place the title window correctly below the icon.

The information below describes how to get the window handle for the small title window, so that the appropriate call to MoveWindow() can be made.

The icon's title window is a window of class #32772. This window is a child of the MDI client window, and its owner is the icon window.

To get the window handle to the appropriate icon title window for a given icon, enumerate all the children of the MDI client window, looking for a window whose owner is the icon.

You can use EnumChildWindows() to loop through all the children of the MDI client window.

You can use GetWindow(..., GW_OWNER) to check the parent of each window.

Additional reference words: 3.00 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrMdi

Use of Allocations w/ cbClsExtra & cbWndExtra in Windows

PSS ID Number: Q11606

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The following is an explanation of the use of the allocations with cbClsExtra and cbWndExtra?:

1. cbClsExtra -- extra bytes to allocate to CLASS data structure in USER.EXE local heap when RegisterClass() is called. Accessed by Get/Set CLASS Word/Long ();.

2. cbWndExtra -- extra bytes to allocate to WND data structure in USER.EXE local heap when CreateWindow() is called. Accessed by Get/Set WND Word/Long ();.

You can use these structures at your discretion and for any purpose you desire.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCls

Use of DLGINCLUDE in Resource Files

PSS ID Number: Q91697

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0

SUMMARY

The dialog editor needs a way to know what include file is associated with a resource file that it opens. Rather than prompt the user for the name of the include file, the name of the include file is embedded in the resource file in most cases.

MORE INFORMATION

Embedding the name of the include file is done with a resource of type RCDATA with the special name DLGINCLUDE. This resource is placed into the .RES file and contains the name of the include file. The dialog editor looks for this resource when it loads a .RES file. If this resource is found, then the include file is opened also; if not, the editor prompts the user for the name of the include file.

In some Windows 3.1 build environments, the dialog editor was used to create dialogs that were placed in more than one .DLG file. These different .DLG files were then included in one .RC file, which was compiled with the resource compiler. Therefore, the resource file gets multiple copies of a RCDATA type resource with the same name, DLGINCLUDE, but the resource compiler and dialog editor don't complain.

In the Win32 SDK, changes were made so that this resource has its own resource type; it was changed from an RCDATA-type resource with the special name, DLGINCLUDE, to a DLGINCLUDE resource type whose name can be specified. The dialog editor would look for resources of the type DLGINCLUDE.

We are being more strict about the need for resources to be unique in the Win32 SDK than the Windows 3.1 SDK. This is good because there was never any guarantee at run time as to which of the two or more resources would be returned by LoadResource().

This means that some applications being ported to Windows NT give an error when their resources are compiled because they have duplicate RCDATA type resources with the same name (DLGINCLUDE). This error is by design. The workaround is straightforward: delete all the DLGINCLUDE RCDATA type resource statements from all the .DLG files.

Finally, because it does not make sense to have the DLGINCLUDE type resources in the executable, the linker will strip them out so that they don't get linked into the EXE.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsDlg

Use of DocumentProperties() vs. ExtDeviceMode()

PSS ID Number: Q92514

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51
    - Microsoft Windows 95 version 4.0

Windows-based applications have used ExtDeviceMode() to retrieve or modify device initialization information for printer drivers. The Win32 API introduces a new function DocumentProperties() that applications can use to configure the settings of the printer.

Note that ExtDeviceMode() calls DocumentProperties(); therefore, it is faster for applications to use DocumentProperties() directly.

Specifying the DM_UPDATE mask allows an application to change printer settings when using DocumentProperties(). Applications should be aware that the GetProcAddress() function is now case sensitive.

Windows-based applications running on Windows NT (WOW) can call ExtDeviceMode(). The spooler's ExtDeviceMode() entry is intended for WOW use.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprint
KBSubcategory: GdiPrn

Use of NULL_PEN, NULL_BRUSH, and HOLLOW_BRUSH

PSS ID Number: Q66532

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

GDI contains several "NULL" stock objects: NULL_BRUSH, HOLLOW_BRUSH, and NULL_PEN. These objects are defined in WINDOWS.H (16-bit SDK) or in WINGDI.H (32-bit SDK). These header files define HOLLOW_BRUSH as NULL_BRUSH, so they are the same objects.

Note that NULL_BRUSH and NULL_PEN are NOT identical to the value NULL. The value NULL is defined as 0 (zero) in WINDOWS.H and is not a valid stock object.

Many GDI functions use the current brush to fill interiors and the current pen to draw lines. In some cases, an application may not want to modify the areas normally affected by the pen or brush. Selecting a NULL_PEN or NULL_BRUSH into the device context tells GDI not to modify the normally affected areas. In short, "NULL_" objects do not draw anything.

For example, the Rectangle() function uses the current brush to fill the interior of the rectangle and the current pen to draw the border. If NULL_PEN is selected into the device context, no border is drawn. If NULL_BRUSH or HOLLOW_BRUSH is selected, the interior of the rectangle is not painted. If both NULL_PEN and NULL_BRUSH are selected, the rectangle will not be drawn.

Additional reference words: 3.00 3.10 3.50 4.00 95 hollow
KBCategory: kbgraphic
KBSubcategory: GdiPnbr

Use of Polygon() Versus PolyPolygon()

PSS ID Number: Q119164

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

Polygon() draws a polygon, while PolyPolygon() draws a series of polygons. Using multiple calls to Polygon() can offer better performance than using a single call to PolyPolygon(); this is because PolyPolygon() does not consider the polygons to be independent, but considers them to be one polygon with multiple disjointed edges. However, there are times when PolyPolygon() is preferable, particularly if the number of polygons is small.

MORE INFORMATION

PolyPolygon() batches polygons in a single call, so there is less call overhead than there is for multiple calls to Polygon(). However, to perform one combined fill, PolyPolygon() has to work with all the edges in all of the polygons simultaneously, resulting in sorting overhead. The overhead involved in sorting becomes quite expensive when there are a lot of polygons, causing a net loss of performance in comparison to Polygon().

GDI batches multiple Polygon() calls to be more efficient. Setting the batch limit higher than the default of 10 with GdiSetBatchLimit() improves performance even further. GDI and some drivers optimize convex polygons, but will only optimize a single polygon drawn with either Polygon() or PolyPolygon().

Because PolyPolygon() treats all edges as part of one big polygon, it also draws every pixel to be filled exactly once; this may be a performance advantage if a lot of overlapping polygons are drawn, because Polygon() draws every pixel in each polygon only once, even where there is an overlap.

PolyPolygon() considers all the polygons when applying the current fill mode, as set by calling SetPolyFillMode(). Consequently, if any polygons overlap, the result of one PolyPolygon() call may be different than the result of the equivalent multiple Polygon() calls. If the polygons overlap and the raster operation takes the destination pixel values into account, or if you want the fill rule to be applied to overlapping areas, then it is preferable to use PolyPolygon().

Additional reference words: 3.10 3.50
KBCategory: kbgraphic
KBSubcategory: GdiDrw

Use Uppercase "K" for Keywords in Windows Help Files

PSS ID Number: Q64050

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

The standard keyword list must be defined by using capital "K" footnotes. Lowercase "k" footnotes may not be used for defining either standard or alternate keyword lists.

MORE INFORMATION

Using lowercase "k" footnotes can result in problems such as the following:

If your application calls WinHelp() using the HELP_KEY option for doing a keyword search and you pass a LPSTR to a keyword defined in a footnote attached to your topic, the Help system displays an "Invalid key word" error message box. For example

```
WinHelp(hWnd,cFileDir,HELP_KEY,(DWORD)(LPSTR)"help");
```

where

```
hWnd     is the handle of the calling window.
cFileDir is the directory path and filename of the .HLP file.
"help"   is the keyword defined in the footnote section of the topic.
```

and the footnote section of the topic is as follows:

```
k sample;help
```

Modifying the footnote for the topic to use an uppercase "K" solves the problem.

```
K sample;help
```

Additional reference words: 3.00 3.10 3.50 4.00 95 key word
KBCategory: kbtool
KBSubcategory: TlsHlp

Using #include Directive with Windows Resource Compiler

PSS ID Number: Q80945

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

SUMMARY

=======

The Windows Resource Compiler supports many standard C language preprocessor directives such as "#define" to define symbolic constants, and "#include" to include header and other resource files.

If an application developed for the Windows environment has more than one resource, each resource can be maintained in a separate file. Then, use the #include directive to direct the Resource Compiler to build all the resources into one output file. Using this technique prevents one resource file from becoming unmanageably large with an overwhelming number of resources.

It is important to note that the Resource Compiler treats files with the .C and .H extensions in a special manner. It assumes that a file with one of these two extensions does not contain resources. When a file has the .C or .H file extension, the Resource Compiler ignores all lines in the file except for preprocessor directives (#define, #include, and so forth). Therefore, a file that contains resources that is included in another resource file should not have the .C or .H file extension.

MORE INFORMATION

================

The following example demonstrates the implications of this situation. The MSG.H file has the following contents:

```
/*
 * This header file defines message IDs for strings in the
 * stringtable resource. All source files can use this header file
 * to reference specific strings.
 */
#define STRING1   1
#define STRING2   2
#define STRING3   3
```

The STRTABLE.RC file has the following contents:

```
/*
 * This file defines the stringtable resource contents for this
 * application. It should be included in the application's resource
 * file. It requires definitions from the MSG.H header file.
```

```
    */
   STRINGTABLE
      {
      STRING1, "This is string 1."
      STRING2, "This is string 2."
      STRING3, "This is string 3."
      }
```

The APP.RC file has the following contents:

```
   /*
    * This is the "main" resource definition file for this
    * application. Among other things, it includes the stringtable
    * resource definition from other header files.
    */

   RESOURCE 1 (MENU)
   RESOURCE 2 (RAW DATA)
   ...

   #include "MSG.H"
   #include "STRTABLE.RC"
```

The Resource Compiler treats both MSG.H and STRTABLE.RC as header files. MSG.H does not include any resources; therefore, it can use the standard .H file extension. However, because STRTABLE.RC includes a resource (a STRINGTABLE), it cannot be named with a .C or .H file extension.

Files that contain resources can have any legal MS-DOS file extension other than .C and .H.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsRc

Using a Dialog Box as the Main Window of an Application

PSS ID Number: Q108936

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Dialog boxes might be used as the main window of an application for several reasons. When an application uses a dialog box as the main window, the following should be taken into consideration while designing such an application:

- The dialog box that acts as the main window of an application can be created without an owner.

- If a modal dialog box is created as a main window, TranslateAccelerator() cannot be used.

- The icon for the dialog box should be drawn manually when the dialog box is minimized.

MORE INFORMATION

You can create a modal or modeless dialog box as the main window of an application. In doing so, there is no need to have an overlapped window that acts as the owner of the dialog box. Memory for edit controls created with the DS_LOCALEDIT flag set, and static controls, will come from the heap represented by the hInstance passed to the CreateDialog() or DialogBox() call.

When modal dialog boxes are chosen for this purpose, and accelerator keys are defined in an application, Windows enters a modal message loop that does not process accelerators, unless a WH_MSGFILTER hook is installed and TranslateAccelerator() is called from the hook callback function.

One easy way to avoid this limitation is to create a modeless dialog box and call TranslateAccelerator() from the main message loop.

The icon for a window is stored in its class information. Because a dialog box is a window of global class that all applications in the system are using, changing the icon for this application will change the icon for all dialog boxes in the system. Below is one workaround for drawing the icon manually when the dialog box is minimized:

Sample Code

```
BOOL WINAPI GenericDlgProc (HWND hwnd, UINT msg,
                            WPARAM wParam,  LPARAM lParam)
 {

   RECT rect ;
   switch (msg) {

       case WM_INITDIALOG:
        hIcon = LoadIcon(); // Load the icon that is to be displayed
                            // when minimized.
        return TRUE ;

       case WM_ERASEBKGND:
            if (IsIconic(hwnd) && hIcon) {
              SendMessage( hwnd, WM_ICONERASEBKGND, wParam, 0L );
              return TRUE;
             }
        break;

       case WM_QUERYDRAGICON:
            return (hIcon);

        case WM_PAINT: {
            PAINTSTRUCT ps;

            BeginPaint( hwnd, &ps );

            if (IsIconic(hwnd))   //*** If iconic, paint the icon.
            {
              if (hIcon) {

                //center the icon correctly...
              GetClientRect(hwnd, &rect) ;
              rect.left = (rect.right - GetSystemMetrics(SM_CXICON)) >> 1;
              rect.top = (rect.bottom - GetSystemMetrics(SM_CYICON)) >> 1;
              DrawIcon( ps.hdc, rect.left, rect.top, hIcon );
                  }
            }
            EndPaint( hwnd, &ps );
        }
        break;
    }
    return FALSE;

} //*** GenericDlgProc
```

The workaround described above assumes that the dialog box belongs to the predefined dialog class. For private dialog box classes, there is no need to manually draw the icon for the dialog box when it is minimized. You can specify the icon while registering the dialog box class. Remember to set the cbWndExtra field to DLGWINDOWEXTRA. When the dialog box is minimized, the icon will be painted automatically.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 DLGMAIN
KBCategory: kbui
KBSubcategory: UsrDlgs

Using a Fixed-Pitch Font in a Dialog Box

PSS ID Number: Q77991

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

To use a fixed-pitch font in a dialog box, during the processing of the dialog box initialization message, send the WM_SETFONT message to each control that will use the fixed font. The following code demonstrates this process:

```
   case WM_INITDIALOG:
     SendDlgItemMessage(hDlg, ID_CONTROL, WM_SETFONT,
           GetStockObject(ANSI_FIXED_FONT), FALSE);
     /*
      * NOTE: This code will specify the fixed font only for the
      * control ID_CONTROL. To specify the fixed font for other
      * controls in the dialog box, additional calls to
      * SendDlgItemMessage() are required.
      */
     break;
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Using a Modeless Dialog Box with No Dialog Function

PSS ID Number: Q72136

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

When creating a modeless dialog box with the default dialog class, an application normally passes a procedure-instance address of the dialog function to the CreateDialog() function. This dialog function processes messages such as WM_INITDIALOG and WM_COMMAND, returning TRUE or FALSE.

It is acceptable to create a modeless dialog box that uses NULL for the lpDialogFunc parameter of CreateDialog(). This type of dialog box is useful when the no controls or other input facilities are required. In this case, using NULL simplifies the programming.

However, the dialog box must be closed through some means other than a push button (for example, via a timer event).

NOTE: A modal dialog box that does not provide a means of closing itself will hang its parent application because control will never return from the DialogBox() function call.

If lpDialogFunc is NULL, no WM_INITDIALOG message will be sent, and DefDlgProc() does not attempt to call a dialog function. Instead, DefDlgProc() handles all messages for the dialog. The application that created the modeless dialog must explicitly call DestroyWindow() to free its system resources.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Using a Mouse with MEP Under Windows NT

PSS ID Number: Q83300

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

The Microsoft Editor (MEP) included with the Win32 Software Development Kit (SDK) can be used with a mouse to position the cursor.

By default, the mouse is not enabled for MEP. It is necessary to add the switch "usemouse:yes" (without the quotation marks) to the TOOLS.INI file under the [m mep] section. The TOOLS.INI file is a text file editable by MEP or Notepad.

To change the position of the cursor, first position the mouse pointer on the new location and then click the left mouse button.

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsMep

Using Built-In Printing Features from a Rich Edit Control

PSS ID Number: Q129860

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY
=======

The Rich Edit control contains built-in printing features that can be used to send formatted text to the printer with minimal effort from the programmer.

MORE INFORMATION
================

Printing from a Rich Edit control involves the use of the standard printing APIs and two Rich Edit control messages, EM_FORMATRANGE and EM_DISPLAYBAND. The EM_FORMATRANGE message can be used by itself or used in combination with the EM_DISPLAYBAND message. Included below at the end of this article is a code sample which demonstrates the usage of these messages.

EM_FORMATRANGE

This message is used to format the text for the printer DC and can optionally send the output to the printer.

The wParam parameter for this message is a Boolean value that indicates whether or not the text should be rendered (printed) to the printer. A zero value only formats the text, while a nonzero value formats the text and renders it to the printer.

The lParam parameter for this message is a pointer to the FORMATRANGE structure. This structure needs to be filled out before sending the message to the control.

FORMATRANGE Members

HDC   hdc - Contains the device context (DC) to render to if the wParam parameter is nonzero. The output is actually sent to this DC.

HDC   hdcTarget - Contains the device context to format for, which is usually the same as the hdc member but can be different. For example, if you create a print preview module, the hdc member is the DC of the window in which the output is viewed, and the hdcTarget member is the DC for the printer.

RECT rc - Contains the area to render to. This member contains the rectangle that the text is formatted to fit in, and subsequently printed in. It also contains the margins, room for headers and footers, and so forth. The rc.bottom member may be changed after the message is sent. If it is changed, it must indicate the largest rectangle that can fit within the bounds of the original rectangle and still contain the specified text without printing partial lines. It may be necessary to reset this value after each page is printed. These dimensions are given in TWIPS.

RECT rcPage - Contains the entire area of the rendering device. This area can be obtained using the GetDeviceCaps() function. These dimensions are given in TWIPS.

CHARRANGE   chrg - Contains the range of characters to be printed. Set chrg.cpMin to 0 and chrg.cpMax to -1 to print all characters.

The return value from EM_FORMATRANGE is the index of the first character on the next page. If you are printing multiple pages, you should set chrg.cpMin to this value before the next EM_FORMATRANGE message is sent.

When printing is complete, this message must be sent to the control with wParam = 0 and lParam = NULL to free the information cache by the control.

EM_DISPLAYBAND

If you use 0 for the wParam parameter in the EM_FORMATRANGE message, then you can use the EM_DISPLAYBAND message to send the output to the printer.

The wParam parameter for this message is not used and should be 0.

The lParam parameter for this message is a pointer to a RECT structure. This RECT structure is the area to display to and is usually the same as the rc member of the FORMATRANGE structure used in the EM_FORMATRANGE message but can be different. For example, the rectangles are not the same if you are printing on a certain portion of a page or built-in margins are being used.

This message should only be used after a previous EM_FORMATRANGE message.

Sample Code

```
void Print(HDC hPrinterDC, HWND hRTFWnd)
   {
   FORMATRANGE fr;
   int         nHorizRes = GetDeviceCaps(hPrinterDC, HORZRES),
               nVertRes = GetDeviceCaps(hPrinterDC, VERTRES),
               nLogPixelsX = GetDeviceCaps(hPrinterDC, LOGPIXELSX),
               nLogPixelsY = GetDeviceCaps(hPrinterDC, LOGPIXELSY);
   LONG        lTextLength;   // Length of document.
   LONG        lTextPrinted;  // Amount of document printed.

   // Ensure the printer DC is in MM_TEXT mode.
```

```
   SetMapMode ( hPrinterDC, MM_TEXT );

   // Rendering to the same DC we are measuring.
   ZeroMemory(&fr, sizeof(fr));
   fr.hdc = fr.hdcTarget = hPrinterDC;

   // Set up the page.
   fr.rcPage.left     = fr.rcPage.top = 0;
   fr.rcPage.right    = (nHorizRes/nLogPixelsX) * 1440;
   fr.rcPage.bottom   = (nVertRes/nLogPixelsY) * 1440;

   // Set up 1" margins all around.
   fr.rc.left   = fr.rcPage.left + 1440;  // 1440 TWIPS = 1 inch.
   fr.rc.top    = fr.rcPage.top + 1440;
   fr.rc.right  = fr.rcPage.right - 1440;
   fr.rc.bottom = fr.rcPage.bottom - 1440;

   // Default the range of text to print as the entire document.
   fr.chrg.cpMin = 0;
   fr.chrg.cpMax = -1;

   // Set up the print job (standard printing stuff here).
   ZeroMemory(&di, sizeof(di));
   di.cbSize = sizeof(DOCINFO);
   if (*szFileName)
      di.lpszDocName = szFileName;
   else
      {
      di.lpszDocName = "(Untitled)";

      // Do not print to file.
      di.lpszOutput = NULL;
      }

   // Start the document.
   StartDoc(hPrinterDC, &di);

   // Find out real size of document in characters.
   lTextLength = SendMessage ( hRTFWnd, WM_GETTEXTLENGTH, 0, 0 );

   do
      {
      // Start the page.
      StartPage(hPrinterDC);

      // Print as much text as can fit on a page. The return value is the
      // index of the first character on the next page. Using TRUE for the
      // wParam parameter causes the text to be printed.

#ifdef USE_BANDING

      lTextPrinted = SendMessage(hRTFWnd,
                                 EM_FORMATRANGE,
                                 FALSE,
                                 (LPARAM)&fr);
```

```
      SendMessage(hRTFWnd, EM_DISPLAYBAND, 0, (LPARAM)&fr.rc);

#else

      lTextPrinted = SendMessage(hRTFWnd,
                                 EM_FORMATRANGE,
                                 TRUE,
                                 (LPARAM)&fr);

#endif

      // Print last page.
      EndPage(hPrinterDC);

      // If there is more text to print, adjust the range of characters to
      // start printing at the first character of the next page.
      if (lTextPrinted < lTextLength)
         {
         fr.chrg.cpMin = lTextPrinted;
         fr.chrg.cpMax = -1;
         }
      }
   while (lTextPrinted < lTextLength);

   // Tell the control to release cached information.
   SendMessage(hRTFWnd, EM_FORMATRANGE, 0, (LPARAM)NULL);

   EndDoc (hPrinterDC);
   }
```

Additional reference words: 1.30 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl W32s

Using cChildren Member of TV_ITEM to Add Speed & Use Less RAM

PSS ID Number: Q131278

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

SUMMARY

The cChildren member of the TV_ITEM structure is used to denote the number of child items associated with a treeview item. When used correctly, the cChildren member helps an application go faster and use less memory.

Applications may specify I_CHILDRENCALLBACK for this member, instead of passing the actual number of child items. To retrieve the actual number of child items associated with a treeview item, I_CHILDRENCALLBACK causes the treeview to send a TVN_GETDISPINFO notification when the item needs to be redrawn.

MORE INFORMATION

In a treeview with the TVS_HASBUTTONS style, specify a nonzero value for the cChildren member to add the appropriate plus or minus (+/-) button (to denote expand or collapse) to the left of the treeview item, without having to insert each of the child items to the treeview. Specifying a zero value indicates that the particular item does not have any child items associated with it.

An application that does not use the cChildren member of the TV_ITEM structure when inserting items to the treeview has to insert each of the child items associated with that treeview item in order for the +/- button to show up.

Using the cChildren member, therefore, helps to speed up an application and reduce memory requirements by allowing the application to fill the tree on demand.

Applications such as the Explorer, WinHelp, and RegEdit, which display huge hierarchical structures, take advantage of this feature by initially inserting only the visible items of the tree. The child items associated with a particular item are not inserted until the user clicks the parent item to expand it. At that point, the treeview's parent window receives a TVN_ITEMEXPANDING notification message, and the application inserts the child items for that parent item:

```
// WM_NOTIFY message handler
LRESULT MsgNotifyTreeView(HWND hwnd,
```

```
                            UINT uMessage,
                            WPARAM wparam,
                            LPARAM lparam)
   {

       LPNMHDR  lpnmhdr = (LPNMHDR)lparam;

       // Just before the parent item gets EXPANDED,
       // add the children.

       if (lpnmhdr->code == TVN_ITEMEXPANDING)
       {
           LPNM_TREEVIEW lpNMTreeView;
           TV_ITEM   tvi;

           lpNMTreeView = (LPNM_TREEVIEW)lparam;
           tvi = lpNMTreeView->itemNew;

           if ((tvi.lParam == PARENT_NODE) &&
               (lpNMTreeView->action == TVE_EXPAND))
           {

               // Fill in the TV_ITEM struct
               // and call TreeView_InsertItem() for each child item.

           }
        }

        return DefWindowProc(hwnd, uMessage, wparam, lparam);
   }
```

When the user clicks the same parent item to collapse it, the application removes all the child items from the tree by using TreeView_Expand (,,TVE_COLLAPSE | TVE_COLLAPSERESET). The TVN_ITEMEXPANDED notification message is a good place to do this:

```
   // WM_NOTIFY message handler
   LRESULT MsgNotifyTreeView(HWND hwnd,
                            UINT uMessage,
                            WPARAM wparam,
                            LPARAM lparam)
   {

       LPNMHDR  lpnmhdr = (LPNMHDR)lparam;

       // Just before the parent item is COLLAPSED,
       // remove the children.

       if (lpnmhdr->code == TVN_ITEMEXPANDED)
       {

           LPNM_TREEVIEW lpNMTreeView;
           TV_ITEM       tvi2;
```

```
            lpNMTreeView = (LPNM_TREEVIEW)lparam;
            tvi2 = lpNMTreeView->itemNew;


            // Do a TVE_COLLAPSERESET on the parent to minimize memory use.

            if ((lpNMTreeView->action == TVE_COLLAPSE) &&
                (tvi2.lParam == PARENT_NODE))
            {
                 TreeView_Expand (ghWndTreeView,
                                  tvi2.hItem,
                                  TVE_COLLAPSE | TVE_COLLAPSERESET);
            }
         }
         return DefWindowProc(hwnd, uMessage, wparam, lparam);
      }
```

Applications that dynamically change the number of child items associated with a particular treeview item may specify I_CHILDRENCALLBACK for the cChildren member of its TV_ITEM structure when it is inserted into the tree. Thereafter, when that treeview item needs to be redrawn, the treeview sends a TVN_GETDISPINFO to its parent window to retrieve the actual number of child items and display the +/- button to the left of the treeview item, as appropriate.

An application that uses I_CHILDRENCALLBACK may process the TVN_GETDISPINFO notification as follows:

```
   // WM_NOTIFY message handler
   LRESULT MsgNotifyTreeView(HWND hwnd,
                             UINT uMessage,
                             WPARAM wparam,
                             LPARAM lparam)
   {

       LPNMHDR  lpnmhdr = (LPNMHDR)lparam;

       if (lpnmhdr->code == TVN_GETDISPINFO)
       {
          TV_DISPINFO FAR *lptvdi;

          lptvdi = (TV_DISPINFO FAR *)lparam;

       if ((lptvdi->item.mask & TVIF_CHILDREN) &&
(lptvdi->item.lParam == PARENT_NODE))
          lptvdi->item.cChildren = 1;

       }

      return DefWindowProc(hwnd, uMessage, wparam, lparam);
   }
```

Additional reference words: 4.00 95 Common Control performance speed up
KBCategory: kbui kbcode
KBSubcategory: UsrCtl

Using Device Contexts Across Threads

PSS ID Number: Q94236

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1, 3.5, and 3.51

A window created with the CS_OWNDC style retains its device context (DC) attributes across GetDC() calls.

However, the DC attributes are not retained if the GetDC() calls are called from different threads. This is by design because DCs are thread-based. In the Win32 user interface, if the calling thread is not the owner of the window, then GetDC() returns a cache DC instead of the owned DC handle.

To save attributes across threads, one must create a routine to initialize DC attributes, which is then called from threads not owning the given window.

Additional reference words: 3.10 3.50
KBCategory: kbgraphic
KBSubcategory: GdiDc

Using Device-Independent Bitmaps and Palettes

PSS ID Number: Q72041

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

The method Windows version 3.x uses to transfer colors from the color table of a device-independent bitmap (DIB) to a device-dependent bitmap (DDB) on a machine that supports palette operations depends on the value of the wUsage parameter specified in calls to the CreateDIBitmap or SetDIBits functions.

Specifying DIB_RGB_COLORS matches the colors in the DIB color table to the logical palette associated with the device context (DC) listed in the function call.

Specifying DIB_PAL_COLORS causes the entries in the DIB color table to not be treated as RGB values; instead, they are treated as word indexes into the logical palette associated with the DC listed in the function call.

To create a device-dependent (displayable) bitmap from a DIB that retains the same colors, follow these five steps:

1. Extract the colors from the DIB header.

2. Use the CreatePalette function to make a logical palette with those colors.

3. Use the SelectPalette function to select the logical palette into a device context.

4. Use the RealizePalette function to map the logical palette into the device context.

5. Call the CreateDIBitmap function using the device context that has the logical palette selected.

Because the CreateDIBitmap function creates a bitmap compatible with the device, to create a device-dependent monochrome bitmap from a DIB, call the CreateBitmap function with the desired width and height, specifying one color plane and one bit per pixel. Then call the SetDIBits function to render the image in the newly created

monochrome bitmap.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiPal

Using Drag-Drop in an Edit Control or a Combo Box

PSS ID Number: Q86724

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In the Microsoft Windows environment, an application can register an edit control or a combo box as a drag-drop client through the DragAcceptFiles function. The application must also subclass the control to process the WM_DROPFILES message that Windows sends when the user drops a file.

MORE INFORMATION

The following seven steps demonstrate how to implement drag-drop in an edit control. The procedure to implement drag-drop in a combo box is identical.

1. Add SHELL.LIB to the list of libraries required to build the file.

2. Add the name of the subclass procedure (MyDragDropProc) to the EXPORTS section of the module definition (DEF) file.

3. Include the SHELLAPI.H file in the application's source code.

4. Declare the following procedure and variables:

```
BOOL FAR PASCAL MyDragDropProc(HWND, unsigned, WORD, LONG);

FARPROC lpfnDragDropProc, lpfnOldEditProc;
char    szTemp64[64];
```

5. Add the following code to the initialization of the dialog box:

```
case WM_INITDIALOG:
   // ... other code

   // ------- edit control section --------
   hWndTemp = GetDlgItem(hDlg, IDD_EDITCONTROL);
   DragAcceptFiles(hWndTemp, TRUE);

   // subclass the drag-drop edit control
   lpfnDragDropProc = MakeProcInstance(MyDragDropProc, hInst);
```

```
         if (lpfnDragDropProc)
            lpfnOldEditProc = SetWindowLong(hWndTemp, GWL_WNDPROC,
                  (DWORD)(FARPROC)lpfnDragDropProc);
         break;
```

6. Write a subclass window procedure for the edit control.

```
      BOOL FAR PASCAL MyDragDropProc(HWND hWnd, unsigned message,
                                     WORD wParam, LONG lParam)
      {
         int wFilesDropped;

         switch (message)
            {
         case WM_DROPFILES:
            // Retrieve number of files dropped
            // To retrieve all files, set iFile parameter
            // to -1 instead of 0
            wFilesDropped = DragQueryFile((HDROP)wParam, 0,
                  (LPSTR)szTemp64, 63);

            if (wFilesDropped)
               {
               // Parse the file path here, if desired
               SendMessage(hWnd, WM_SETTEXT, 0, (LPSTR)szTemp64);
               }
            else
               MessageBeep(0);

            DragFinish((HDROP)wParam);
            break;

         default:
            return CallWindowProc(lpfnOldEditProc, hWnd, message,
                  wParam, lParam);
            break;
         }
         return TRUE;
      }
```

7. After the completion of the dialog box procedure, free the edit control subclass procedure.

```
      if (lpfnDragDropProc)
         FreeProcInstance(lpfnDragDropProc);
```

Additional reference words: 3.10 3.50 3.51 4.00 95 combobox
KBCategory: kbui
KBSubcategory: UsrDnd

Using DWL_USER to Access Extra Bytes in a Dialog Box

PSS ID Number: Q88358

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Windows extra bytes are used to store private information specific to an instance of a window. For dialog boxes, these extra bytes are already allocated by the dialog manager. The offset to the extra byte location is called DWL_USER.

DWL_USER is the 8th byte offset of the dialog extra bytes. The programmer has 4 bytes (a long) available from this offset for personal use.

CAUTION: DO NOT use more than 4 bytes of these extra bytes, as the rest of them are used by the dialog manager.

Example

```
   DWORD dwNumber = 10;
            .
            .
            .
            .
   case WM_INITDIALOG:
           SetWindowLong(hWnd,DWL_USER,dwNumber); // Store value 10 at
                                                  // byte offset 8
           dwNumber = GetWindowLong(hWnd,DWL_USER); // Retrieve the value
```

NOTE: GetWindowWord and SetWindowWord could be used instead.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Using ENTER Key from Edit Controls in a Dialog Box

PSS ID Number: Q102589

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Windows-based applications often display data-entry dialog boxes to request information from users. These dialog boxes may contain several edit controls and two command (push) buttons, labeled OK and CANCEL. An example of a data-entry dialog box is one that requests personal information, such as social security number, address, identification (ID) number, date/time, and so on, from users. Each of these items is entered into an edit control.

By default, the TAB key is used in a dialog box to shift focus between edit controls. As a common user-interface, however, one could also use the ENTER (RETURN) key to move between the edit controls (for example, after the user enters a piece of information, pressing ENTER moves the focus to the next field).

There are a few ways to enable the use of the ENTER key to move between edit controls. One method is to make use of WM_COMMAND and the notification messages that come with it in the dialog box for edit controls and buttons. Another method involves subclassing the edit controls. A third involves using App Studio and Class Wizard and creating a new dialog box member function.

MORE INFORMATION

Method I: (WM_COMMAND)

This method is based on the following behavior of dialog boxes (Dialog Manager) and focus handling in Windows.

If a dialog box or one of its controls currently has the input focus, then pressing the ENTER key causes Windows to send a WM_COMMAND message with the idItem (wParam) parameter set to the ID of the default command button. If the dialog box does not have a default command button, then the idItem parameter is set to IDOK by default.

When an application receives the WM_COMMAND message with idItem set to the ID of the default command button, the focus remains with the control that had the focus before the ENTER key was pressed. Calling

GetFocus() at this point returns the handle of the control that had the focus before the ENTER key was pressed. The application can check this control handle and determine whether it belongs to any of the edit controls in the dialog box. If it does, then the user was entering data into one of the edit controls and after doing so, pressed ENTER. At this point, the application can send the WM_NEXTDLGCTL message to the dialog box to move the focus to the next control.

However, if the focus was with one of the command buttons (CANCEL or OK), then GetFocus() returns a button control handle, at which point one can dismiss the dialog box. The pseudo code for this logic resembles the following in the application's dialog box procedure:

```
     case WM_COMMAND:

     if(wParam=IDOFDEFBUTTON || IDOK) {
          // User has hit the ENTER key.

          hwndTest = GetFocus() ;
          retVal = TesthWnd(hWndTest) ;

          //Where retVal is a boolean variable that indicates whether
        //the hwndTest is the handle of one of the edit controls.

          if(hwndTest) {
         //Focus is with an edit control, so do not close the dialog.
            //Move focus to the next control in the dialog.

         PostMessage(hDlg, WM_NEXTDLGCTL, 0, 0L) ;
         return TRUE ;
          }
          else {
            //Focus is with the default button, so close the dialog.
            EndDialog(hDlg, TRUE) ;
            return FALSE ;
          }
        }
     break ;
```

Method II

This method involves subclassing/superclassing the edit control in the dialog box. Once the edit controls are subclassed or superclassed, all keyboard input is sent the subclass/superclass procedure of the edit control that currently has input focus, regardless of whether or not the dialog box has a default command button. The application can trap the key down (or char) messages, look for the ENTER key, and do the processing accordingly. The following is a sample subclass procedure that looks for the ENTER key:

```
//*---------------------------------------------------------------------
//| Title:
//|     SubClassProc
```

```
//|
//| Parameters:
//|     hWnd            - Handle to the message's destination window
//|     wMessage        - Message number of the current message
//|     wParam          - Additional info associated with the message
//|     lParam          - Additional info associated with the message
//|
//| Purpose:
//|     This is the window procedure used to subclass the edit control.
//*---------------------------------------------------------------------

long FAR PASCAL SubProc(HWND hWnd, WORD wMessage,WORD wParam,LONG lParam)
{

    switch (wMessage)
     {

          case WM_GETDLGCODE:
            return (DLGC_WANTALLKEYS |
                    CallWindowProc(lpOldProc, hWnd, wMessage,
                                   wParam, lParam));

          case WM_CHAR:
          //Process this message to avoid message beeps.
         if ((wParam == VK_RETURN) || (wParam == VK_TAB))
           return 0;
            else
           return (CallWindowProc(lpOldProc, hWnd,
                                  wMessage, wParam, lParam));

       case WM_KEYDOWN:
            if ((wParam == VK_RETURN) || (wParam == VK_TAB)) {
              PostMessage (ghDlg, WM_NEXTDLGCTL, 0, 0L);
              return FALSE;
            }

         return (CallWindowProc(lpOldProc, hWnd, wMessage,
                                wParam, lParam));
         break ;

       default:
            break;

     } /* end switch */
```

Method 3

This method involves using App Studio and ClassWizard and creating a new dialog box member function.

This method will allow a user to press the ENTER key and have the focus advance to the next edit control. If the focus is currently on the last edit control in the dialog box, the focus will advance to the first edit control.

First, use App Studio to change the ID of the OK button of the dialog box. The default behavior of App Studio is to give the OK button the ID IDOK. The OK button's ID should be changed to another value, such as IDC_OK. Also, change the properties of the OK button so that it is not a default pushbutton.

Next, use ClassWizard to create a new dialog box member funciton. Name the new member function something like OnClickedOK. This function should be tied to the BN_CLICKED message from the IDC_OK control.

Once this is done, write the body of the OnClickedOK function. You should put the code that you would normally put in the OnOK function into the new OnClickedOK function, including a class's OnOK function.

Add the following prototype to the header file for the dialog box:

```
   protected:
      virtual void OnOK();
```

Add an OnOK function to the dialog box and code is as demonstrated below:

```
void CMyDialog::OnOK()
{
     CWnd* pwndCtrl = GetFocus();
     CWnd* pwndCtrlNext = pwndCtrl;
     int ctrl_ID = pwndCtrl->GetDlgCtrlID();

     switch (ctrl_ID) {
         case IDC_EDIT1:
             pwndCtrlNext = GetDlgItem(IDC_EDIT2);
             break;
         case IDC_EDIT2:
             pwndCtrlNext = GetDlgItem(IDC_EDIT3);
             break;
         case IDC_EDIT3:
             pwndCtrlNext = GetDlgItem(IDC_EDIT4);
             break;
         case IDC_EDIT4:
             pwndCtrlNext = GetDlgItem(IDC_EDIT1);
             break;
         case IDOK:
             CDialog::OnOK();
             break;
         default:
             break;
     }
     pwndCtrlNext->SetFoucs();
}
```

Additional reference words: 3.10 3.50 3.51 4.00 95 push RETURN keydown
KBCategory: kbui
KBSubcategory: UsrDlgs

Using Extra Fields in Window Class Structure

PSS ID Number: Q10841

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

In order to generate several child windows of the same class, each having its own set of static variables and independent of the sets of the variables in the sibling windows, you need to use the cbWndExtra field in WNDCLASS, the window-class data structure, when registering a window; then, use SetWindowWord() (or Long) and GetWindowWord() (or Long). These functions will either get or set additional information about the window identified by hWnd.

Use positive offsets as indexes to access any additional bytes that were allocated when the window class structure was created, starting at zero for the first byte of the extra space. Similarly, if you want to refer to bytes already defined by Windows within the structure, use offsets defined with the GWW and GWL prefixes.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrCls

Using FileOpen Common Dialog w/ OFN_ALLOWMULTIPLESELECT Style

PSS ID Number: Q130761

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

SUMMARY

=======

This article covers the format of file names returned by the FileOpen common dialog with the OFN_ALLOWMULTISELECT style.

The FileOpen common dialog can be used to specify the location (drive and directory) and name of a file or a set of files. One of the flags needed to provide mulitple-file selection from the FileOpen common dialog is the OFN_ALLOWMULTISELECT flag. When this flag is used, and the user makes a valid selection, the file or files chosen by the user are returned in the lpstrFile member of the OPENFILENAME structure. The format of the string returned in the lpstrFile member depends on how many files (single or multiple) the user selected. This article assumes that the OFN_EXPLORER Style is set for the file open dialog.

MORE INFORMATION

================

If multiple files were selected, the string is of this form:

   Drive: \Directory Name\0FileName 1\0FileName 2\0FileName n\0\0.

The Directory Name is listed first. Then each file that was selected is listed with a terminating NULL Character, except for the last filename, which is terminated with two NULL characters. The two NULL characters signal the end of the string.

If a single file was selected, the string is of this form:

   Drive: \Directory Name\FileName\0\0.

The Directory Name in this case is not terminated by a NULL character, and the file name is terminated with two NULL characters.

Applications must parse the string returned in the lsptrFile member. In doing so, they should make provisions in the parsing code to have a case where the user can make a single selection (even though the OFN_ALLOWMULTISELECT flag is set) or multiple selections.

NOTE: When using OFN_ALLOWMULTISELECT under Windows 95, you need to use the OFN_EXPLORER flag to get the explorer style dialog and NULL terminated strings. If you don't use OFN_EXPLORER with OFN_ALLOWMULTISELECT, you

get the old style dialog and space-delimited strings.

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Using GDI-Synthesized Italic Fonts

PSS ID Number: Q74467

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

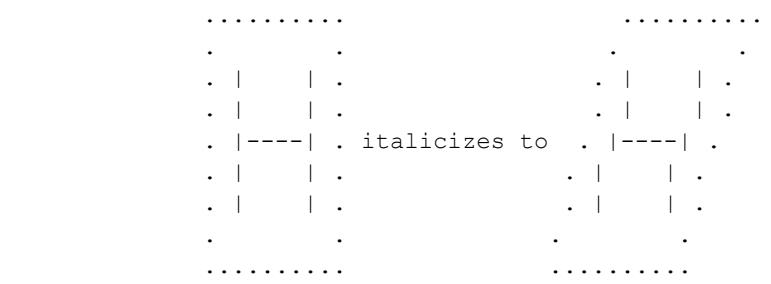
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

In the Microsoft Windows graphical environment, when an application uses an italic font synthesized by the graphics device interface (GDI), each character and its whole character cell are "sheared," or slanted, to the right, which can cause some unexpected results.

MORE INFORMATION
================

The capital H in the example below illustrates how GDI synthesizes an italic font:



Note two items in this case:

1. If the text background color is changed so that it does not match the window background color, the text background color occupies the sheared character cell (in other words, it is also slanted). Gaps occur in the background where normal text is adjacent to italic text.

2. The italic character is farther to the right in relation to the lower-left corner of the character cell than is the normal character. Therefore, if normal and italic text start at the same x coordinate on different lines, the italic text appears farther to the right.

To determine the number of units by which the character cell is sheared, call the GetTextMetrics function to fill a TEXTMETRIC data

structure with information about the font. The tmOverhang member describes the amount of shear.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiFnt

Using GetDIBits() for Retrieving Bitmap Information

PSS ID Number: Q85846

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5
  - Microsoft Windows 95 version 4.0

SUMMARY

When saving a bitmap in .DIB file format, the GDI function is used to retrieve the bitmap information. The general use of this function and the techniques for saving a bitmap in .DIB format are largely unchanged; however, this article provides more details on the use of the Win32 API version of the GetDIBits() function.

MORE INFORMATION

The function can be used to retrieve the following information:

- Data in the BitmapInfoHeader (no color table and no bits)
- Data in the BitmapInfoHeader and the color table (no bits)
- All the data (BitmapInfoHeader, color table, and the bits)

The fifth and the sixth parameters of the function are used to tell the graphics engine exactly what the application wants it to return. If the fifth parameter is NULL, then no bits will be returned. If the biBitCount is 0 (zero) in the sixth parameter, then no color table will be returned. In addition, the biSize field of the BitmapInfoHeader must be set to either the size of BitmapInfoHeader or BitmapCoreHeader for the function to work properly.

Refer to the SAVEBMP.C file in the MANDEL sample for details. This sample is included with the Win32 SDK.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiBmp

Using GetForegroundWindow() When Desktop Is Not Active

PSS ID Number: Q118624

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

GetForegroundWindow() is documented to return the handle of the foreground window, that is, the window that the user is currently working with. The proper handle is returned when the desktop that the application is running on is active; however, when another desktop is active, GetForegroundWindow() returns NULL.

This is expected behavior. There is no way to get the active window in your own desktop while another desktop is active.

The application desktop is one desktop. Other desktops include the logon and screen saver desktops. If GetForegroundWindow() returned a handle to the logon dialog box, it would be possible to create an application that could get user passwords. This would violate Windows NT security.

For this reason, it is not possible to create screen savers that melt or drop out.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrWndw

Using GetUpdateRgn()

PSS ID Number: Q99047

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

The documentation provided with the Microsoft Windows Software Development Kit (SDK) for Microsoft Windows version 3.1 does not address all the issues surrounding the use of the GetUpdateRgn() function. This article complements the documentation of GetUpdateRgn().

MORE INFORMATION
================

The handle to the region (hRgn) does not have to be selected into a device context (DC) to be able to use it with GetUpdateRgn(). Even if the hRgn is selected as the clipping region for a DC, Windows only makes a copy of the hRgn for the hDC instead of using the hRgn directly.

When hRgn is used with GetUpdateRgn(), Windows will ignore any existing region in hRgn. It will replace any existing region with the current update region of the window.

Calling GetUpdateRgn() soon after BeginPaint() always yields an empty region because BeginPaint() validates the update region.

A typical Windows query function (functions that typically begin with Get and Is) does not initiate any action. GetUpdateRgn() is not a typical function in this respect. The third parameter, fErase, works as advertised to initiate a repaint on request.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrPnt

Using GMEM_DDESHARE in Win32 Programming

PSS ID Number: Q99114

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

The GMEM_DDESHARE flag remains a legitimate value for GlobalAlloc(). This flag can be used to indicate that the memory will be used for one of the following so that the system can optimize the allocation for these special needs:

   DDE
   OLE 1.0
   Clipboard operations

However, GlobalAlloc(GMEM_DDESHARE, ...) cannot be used to allocate a block of memory that can be shared between processes. This flag was never intended for this purpose, even under Windows versions 3.0 and 3.1 (3.x). GlobalAlloc(GMEM_DDESHARE, ...) works in this case because all Windows-based applications share the same address space; this is not the case under Windows NT.

All allocations of global shared memory can be used within the process that they are allocated in, but another mechanism is required to share memory between processes.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseMm

Using Graphics Within a Help File

PSS ID Number: Q90291

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, versions 3.51 and 4.0

SUMMARY

This article explains how to use graphics in a Help file with both the help compiler 3.1 (HC30.EXE, HC31.EXE, and HCP.EXE) and the help compiler 4.0 (HCW.EXE).

MORE INFORMATION

TYPES OF GRAPHICS

With the Help Compiler, four types of graphics can be displayed within help topics: bitmaps, metafiles, segmented hypergraphics, and multiple resolution bitmaps. With HC30 and HC31, these graphics are limited to 16 colors but it is possible to use embedded windows to create a 256-color bitmap. HCW is capable of displaying 16 million colors.

The following section discusses the details of the graphics formats listed above, and provides details on their advantages and disadvantages.

Bitmaps

A bitmap is an image that is described by a matrix of memory bits that, when copied to a device, define the color and pattern of a corresponding matrix of pixels on the display surface of the device. The advantage to using a bitmap is that drawing a bitmap is very fast. The disadvantage is that the size of a bitmap is very large. Bitmaps can be created with a graphics editor, such as Paintbrush.

Metafiles

A metafile is a collection of GDI commands that creates desired text or images. There are two advantages to using metafiles: the size of the metafile is small, and metafiles are less device-dependent than bitmaps. The disadvantage of using a metafile is that it takes a long time to draw one.

Segmented Hypergraphics

A segmented hypergraphic is a graphic that has hot spots defined in various

regions of the graphic. Clicking hot spots either executes a macro or jumps to a context string. To make a segmented hypergraphic, use the segmented hypergraphic (hot spot) editor (SHED.EXE) included with the Windows 3.1 SDK.

Multiple Resolution Bitmaps

A multiple resolution bitmap is a single bitmap file that contains one or more bitmaps that have been marked for use with specific displays. The advantages of multiple resolution bitmaps are:

1. Bitmaps are prevented from appearing too big or too small on different resolutions.

2. Bitmaps are prevented from looking stretched or compressed from display to display.

3. Colors are mapped correctly on different displays.

The disadvantage of multiple resolution bitmaps is that the files are large. multiple resolution bitmaps can be created from bitmap files with the multiple resolution bitmap compiler, MRBC.

PLACING GRAPHICS

Direct Pasting

Bitmaps and metafiles can be pasted directly from the clipboard into an RTF source file. This allows the help author to see what the topic will look like while it is being edited. There are several disadvantages to this method. The first disadvantage only applies to HC30 and HC31. It is that any graphic pasted directly into a topic is limited to 64K. This is the result of the help compiler's 64K per paragraph limit when processing RTF source files. The second disadvantage applies to all the help compilers. It is size. If the same graphic is used multiple times within the same source file, then a copy of the graphic is made each time it is placed within the source.

By Reference

All of the graphics can be placed "by reference." To insert a graphic by reference, the help author must type {bm? graphic.ext} where bm? is bml, bmr, or bmc and graphic.ext is the name of the graphic file that the author wants to have placed in the help topic. The bml command is used for a left-justified graphic, the bmr command is used for a right justified graphic, and the bmc command is used for a character justified graphic (that is, the graphic is inserted into the paragraph as if it were a character).

One of the advantages of placing a graphic by reference is that it lifts the 64K limit on a graphic. Also, a graphic placed by reference

is actually just a "pointer" to the real graphic. Therefore, if the same graphic is used multiple times, it is only "stored" once within the .HLP file.

The disadvantage of placing graphics by reference is that the help author does not see how the topic will appear while in the RTF editor. The bitmap files inserted by reference must be found in the directory specified by either the ROOT or BMROOT settings in the [OPTIONS] section of the help project file. If the bitmap is not located in one of these directories, then the file must be listed in the [BITMAPS] section of the project file, so the help compiler can locate the bitmap.

Hot Spots and Pop Ups

When placing a graphic into an RTF source file, it can be turned into a hot spot or pop up, similar to other text. Just select the graphic and turn on the double or single underline attribute followed immediately by the hidden text for the context string or macro.

Additional reference words: 3.10 4.00 95 WinHelp
KBCategory: kbtool
KBSubcategory: TlsHlp

Using Network DDE Under Win32s

PSS ID Number: Q125475

The information in this article applies to:

- Microsoft Win32s, version 1.2

Network Dynamic Data Exchange (NetDDE) has limited support in Win32s. You can use DDE across the network, however the NDde APIs are not supported under Win32s and will not be supported in the future.

The NDde APIs, such as NDdeShareAdd(), are used to create the NetDDE shares, not for the actual communication. Therefore, to use NetDDE with Win32s, manually create the shares with the DDESHARE utility included in the Windows for Workgroups Resource Kit or by thunking to the 16-bit NDde APIs. Then communicate through DDE or DDEML.

Additional reference words: 1.20
KBCategory: kbprg kbnetwork
KBSubcategory: W32s

Using NTFS Alternate Data Streams

PSS ID Number: Q105763

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

The documentation for the NTFS file system states that NTFS supports multiple streams of data; however, the documentation does not address the syntax for the streams themselves.

The Windows NT Resource Kit documents the stream syntax as follows:

```
filename:stream
```

Alternate data streams are strictly a feature of the NTFS file system and may not be supported in future file systems. However, NTFS will be supported in future versions of Windows NT.

Future file systems will support a model based on OLE 2.0 structured storage (IStream and IStorage). By using OLE 2.0, an application can support multiple streams on any file system and all supported operating systems (Windows, Macintosh, Windows NT, and Win32s), not just Windows NT.

MORE INFORMATION

================

The following sample code demonstrates NTFS streams:

```
#include <windows.h>
#include <stdio.h>

void main( )
{
   HANDLE hFile, hStream;
   DWORD dwRet;

   hFile = CreateFile( "testfile",
                       GENERIC_WRITE,
                   FILE_SHARE_WRITE,
                               NULL,
                         OPEN_ALWAYS,
                                  0,
                               NULL );
   if( hFile == INVALID_HANDLE_VALUE )
      printf( "Cannot open testfile\n" );
   else
      WriteFile( hFile, "This is testfile", 16, &dwRet, NULL );
```

```
   hStream = CreateFile( "testfile:stream",
                              GENERIC_WRITE,
                           FILE_SHARE_WRITE,
                                       NULL,
                                OPEN_ALWAYS,
                                         0,
                                       NULL );
   if( hStream == INVALID_HANDLE_VALUE )
      printf( "Cannot open testfile:stream\n" );
   else
      WriteFile( hStream, "This is testfile:stream", 23, &dwRet, NULL );
}
```

The file size obtained in a directory listing is 16, because you are looking only at "testfile", and therefore

```
   type testfile
```

produces the following:

```
   This is testfile
```

However

```
   type testfile:stream
```

produces the following:

```
   The filename syntax is incorrect
```

In order to view what is in testfile:stream, use:

```
   more < testfile:stream
```

-or-

```
   mep testfile:stream
```

where "mep" is the Microsoft Editor available in Win32 SDK.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseFileio

Using One IsDialogMessage() Call for Many Modeless Dialogs

PSS ID Number: Q71450

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In the Windows environment, an application can implement more than one modeless dialog box with a single call to the IsDialogMessage() function. This can be done by using the following three-step method:

1. Maintain the window handle to the currently active modeless dialog box in a global variable.

2. Pass the global variable as the hDlg parameter to the IsDialogMessage() function, which is normally called from the application's main message loop.

3. Update the global variable whenever a modeless dialog box's window procedure receives a WM_ACTIVATE message, as follows:

   - If the dialog is losing activation (wParam is 0), set the global variable to NULL.

   - If the dialog is becoming active (wParam is 1 or 2), set the global variable to the dialog's window handle.

MORE INFORMATION

The information below demonstrates how to implement this technique.

1. Declare a global variable for the modeless dialog box's window handle.

```
HWND hDlgCurrent = NULL;
```

2. In the application's main message loop, add a call to the IsDialogMessage() function.

```
while (GetMessage(&msg, NULL, 0, 0))
   {
   if (NULL == hDlgCurrent || !IsDialogMessage(hDlgCurrent, &msg))
      {
      TranslateMessage(&msg);
```

```
         DispatchMessage(&msg);
         }
      }
```

3. In the modeless dialog box's window procedure, process the WM_ACTIVATE message.

```
   switch (message)
      {
      case WM_ACTIVATE:
         if (0 == wParam)             // becoming inactive
            hDlgCurrent = NULL;
         else                         // becoming active
            hDlgCurrent = hDlg;

         return FALSE;
      }
```

For more information on the WM_ACTIVATE message, see page 6-47 in "Microsoft Windows Software Development Kit Reference Volume 1" for the Windows SDK version 3.0 and page 87 of "Programmer's Reference, Volume 3: Messages, Structures, and Macros" for the Windows SDK version 3.1.

For details on the IsDialogMessage() function, see page 4-266 in "Windows Software Development Kit Reference Volume 1" for the Windows SDK version 3.0 and page 553 of "Programmer's Reference, Volume 2: Functions" for the Windows SDK version 3.1.

For details on using a modeless dialog box in an application for the Windows environment, see Chapter 10 of "Programming Windows," second edition, (Microsoft Press) written by Charles Petzold.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Using Printer Escapes w/PS Printers on Windows NT & Win32s

PSS ID Number: Q124135

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Win32s versions 1.1, 1.15, 1.15a, and 1.2

SUMMARY
=======

To identify and print to PostScript printers from a Win32-based application under Windows NT and under Win32s, you need to special-case your code. This is because the printer drivers respond to different Printer Escapes under Windows NT and Windows/Win32s.

This article discusses how to identify and print to PostScript printers on both Windows NT and Win32s.

MORE INFORMATION
================

Identification

To identify the printer as a PostScript printer, use this code:

```
   int gPrCode = 0;  // Set according to platform.

   if( Win32s ) // Using the Win16 driver.
   {
      gPrCode = PASSTHROUGH;
      if((Escape(printerIC, GETTECHNOLOGY, NULL, NULL, (LPSTR)szTech) &&
         !lstrcmp(szTech, "PostScript")) &&
         Escape(printerIC, QUERYESCSUPPORT, sizeof(int),
            (LPSTR)gPrCode, NULL )
      {
      // The printer is PostScript.
         ...
      }
   }
   else // Using Win32 driver under Windows NT.
   {
      gPrCode = POSTSCRIPT_PASSTHROUGH; // Fails with Win16 driver
      if( Escape(printerIC, QUERYESCSUPPORT, sizeof(int), (LPSTR)gPrCode,
          NULL))
      {
      // The printer is PostScript.
         ...
      }
   }
```

Printing

To send PostScript data to the printer on either platform, use this code:

```
// Assuming a buffer, szPSBuf, of max size MAX_PSBUF containing
// nPSData bytes of PostScript data.

   char szBuf[MAX_PSBUF+sizeof(short)];

// Store length in buffer.
   *((short *)szBuf) = nPSData;

// Store data in buffer.
   memcpy( (char *)szBuf + sizeof(short), szPSBuf, nPSData );

// Note that gPrCode (set when identifying the printer) depends on
// the platform.
   Escape( printerDC, gPrCode, (int) nPSData, szBuf, NULL );
```

However, your output may appear scaled or translated incorrectly or data may be transformed off the page under Win32s.

The origin and scale for Windows printer drivers is not the PostScript default (bottom left/72 dpi) but is instead at the upper left and at the device scale(300 dpi). Therefore, before sending data to the printer, you may need to send a couple of PostScript commands to scale or translate the matrix. For example, for scaling, send the following escape to scale the PostScript transform to 72 dpi:

```
   xres = GetDeviceCaps(printerDC, LOGPIXELSX);
   yres = GetDeviceCaps(printerDC, LOGPIXELSY);

// Two leading spaces for the following operation.
   wsprintf(szBuf, "  %d 72 div %d 72 div scale\n", xres, yres);

// Put actual size into buffer
   *((short *)szBuf) = strlen(szBuf)-2;
   Escape( printerDC, gPrCode, strlen(szBuf)-2, szBuf, NULL );
```

Additional reference words: 1.10 1.20 3.10 3.50
KBCategory: kbprint kbcode
KBSubcategory: GdiPrn W32s

Using Private Templates with Common Dialogs

PSS ID Number: Q74609

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.5

SUMMARY
=======

An application which uses the common dialog library (COMMDLG.DLL) can provide its own dialog resource template to be used instead of the standard template. In this way, the application can include private dialog items specific to its needs without losing the benefits of using the COMMDLG's dialog handling.

MORE INFORMATION
================

Each common dialog data structure contains an lpTemplateName element. (Note that the Print Dialog structure contains two such elements, each with a distinct name -- see specifics of PrintDlg for details.) This element points to a null-terminated string that names the dialog box template resource to be substituted for the standard dialog template. If the dialog resource is numbered, the application can use the MAKEINTRESOURCE macro to convert the number into a pointer to a string. Alternatively, the application can choose to pass a handle to a preloaded dialog template. The Flags element of the dialog data structure must be set to indicate which method is being used.

After loading the application's dialog template, the common dialog DLL initializes the dialog items as it would for the standard template. This leads to an important point: all dialog items in the standard template must also exist in the application's private template. Note that the items do not have to be enabled or visible -- they just have to exist.

Once the DLL has finished handling the WM_INITDIALOG message, it passes that message on to the application's dialog hook function. The hook function handles WM_INITDIALOG by initializing the application's private dialog items. It can also disable and hide any items from the standard template that the application does not want to use.

The hook function should process messages and notifications concerning the private dialog items.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrCmnDlg

Using Quoted Strings with Profile String Functions

PSS ID Number: Q69752

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Microsoft Windows provides profile files which are a mechanism for an application to store configuration about itself. The WIN.INI file is the system profile file in which Windows stores configuration information about itself. In versions of Windows prior to version 3.0, applications also stored configuration information in the WIN.INI file. Windows 3.0 introduced private profile files, which can store application-specific information.

An application can retrieve information from a profile file by calling the GetProfileString or GetPrivateProfileString function. If the profile file associates the specified lpKeyName value with a string that is delimited by quotation marks, Windows discards the quotation marks when it copies the associated string into the application-provided buffer.

For example, if the following entry appears in the profile file:

```
   [application name]           [application name]
   keyname = 'string'     or     keyname = "string"
```

The GetPrivateProfileString and GetProfileString functions read the string value and discard the quotation marks.

MORE INFORMATION

This behavior allows spaces to be put into a string. For example, the profile entry

```
   keyname = string
```

returns the string without a leading space, whereas

```
   keyname = ' string'     or     keyname = " string"
```

returns the string with a leading space.

Doubling quotation marks includes quotation marks in the string. For

example:

```
   keyname = ''string''    or     keyname = ""string""
```

returns the string with its quotation marks -- 'string' or "string".

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrIni

Using ReadFile() and WriteFile() on Socket Descriptors

PSS ID Number: Q104536

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

Socket handles for Windows NT sockets are object handles. For example, you can pass a socket handle in the appropriate state to the ReadFile(), ReadFileEx(), WriteFile(), or WriteFileEx() application programming interface (API) to receive and send data. The socket descriptor passed to the file APIs must be a connected, TCP descriptor.

NOTE: There is no way to specify send and receive out-of-band data.

To use a Windows Sockets handle, the ReadFile() and WriteFile() APIs must use asynchronous access. That is, you must specify the overlapped parameter in the call to ReadFile() and WriteFile(). This will allow you to be notified when the I/O has completed.

This functionality is based upon the implementation of Windows Sockets and may not be available to all implementations. For example, although this works in the Win32 subsystem, it is not supported under Win32s.

Additional reference words: 3.10 3.50
KBCategory: kbnetwork
KBSubcategory: NtwkWinsock

Using RLE Bitmaps for Animation Applications In Windows

PSS ID Number: Q75214

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

Animation that uses a "brute-force" animation scheme, displaying successive complete bitmaps, can be slow and choppy. The alternative approach, displaying an initial frame and then modifying the elements that change, can save a great deal of time, memory, and disk space.

MORE INFORMATION
================

Animation applications typically use a very simple algorithm to display a sequence of bitmaps, or frames, one right over another, on the same display surface. When the application stores an entire frame in the device-independent bitmap (DIB) format, it suffers from three particular bottlenecks:

1. With large frames or long sequences, the storage file becomes quite large.

2. Loading an entire sequence requires a large amount of memory.

3. Available Windows functions, such as BitBlt or SetDIBitsToDevice, may be too slow to animate the sequence smoothly.

For an animation sequence of bitmaps with a consistent set of colors, preprocessing the animation sequence and storing it in the run-length-encoded (RLE) format enables an application to run faster with a smaller storage file and memory requirement. Note that this RLE file is different from an RLE compressed DIB. This RLE format consists of several frames that are compressed according to the differences between frames. This process is straightforward and consists of five steps:

1. Select the first frame in the sequence and store it in the DIB format.

2. Compare the DIB bitmaps of consecutive frames. Due to the nature of animation sequences, the number of differences is often quite small compared to the size of the entire frame.

3. Encode the set of changed pixels into RLE format. The encoded frame will contain information only on the pixels that change, the delta records will skip the unchanged pixels. The RLE bitmap is stored instead of the latter frame. For example, the RLE-encoded difference between the first and second frames is stored as the second frame.

4. When the entire sequence is preprocessed, bring each frame into the system as a BITMAPINFO structure and stream of bits. Since only the first frame is in the DIB format, the memory requirement is quite low. Moreover, frames that contain an identical set of colors can share the BITMAPINFO structure, only the biCompression and biSizeImage fields must be changed.

5. At display time, load the first frame as a DIB. Then use the SetDIBitsToDevice function to display subsequent frames in the sequence. Because this function requires much less information, the sequence can be animated much more quickly and smoothly.

Additional reference words: 3.00 3.10 3.50 4.00 95 rle
KBCategory: kbgraphic
KBSubcategory: GdiBmp

Using RPC Callback Functions

PSS ID Number: Q96781

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SUMMARY
=======

The standard remote procedure call (RPC) model has a server containing one or more exported function calls, and a client, which calls the server's exported functions. However, Microsoft's implementation of RPC defines callbacks as a special interface definition language (IDL) attribute allowing a server to call a client function.

Callbacks can be used only in the context of a server call. Thus, a server may call a client's callback function only when the server is performing a client's remote procedure call (before it returns from processing). For example:

```
               CLIENT                SERVER
               ------                ------

       Client makes RPC call. --->
                                <--- Server calls callback procedure.
Client returns from callback. --->
                                <--- Server calls callback procedure.
Client returns from callback. --->
                                <--- Server returns from original RPC call.
```

MORE INFORMATION
================

Callbacks are declared in the RPC .IDL file and defined in the source of the client. The following demonstrates how callbacks are declared and defined:

```
[ SAMPLE.IDL ]
[
   uuid(9FEE4F51-0396-101A-AE4F-08002B2D0065),
   version(1.0),
   pointer_default( unique )
]

{
 void RPCProc( [in, string] unsigned char *pszStr );
 [callback] void CallbackProc([in,string] unsigned char *pszStr);
}


[ SAMPLEC.C (Client)]
```

```
/*
   Callback RPC call (initiated from server, executed on client).
*/
void CallbackProc( unsigned char *pszString )
{
  printf("Call from server, printed on client: %s", pszStr );
}


[ SAMPLES.C (Server)]
/*
   "Standard" RPC call (initiated from client, executed on server).
   Makes a call to client callback procedure, CallbackProc().
*/
void RPCProc( unsigned char *pszStr )
{
  printf("About to call Callback() client function.."
  CallbackProc( pszStr );
  printf("Called callback function.");
}
```

In the makefile for the sample, the "-ms_ext" switch must be used for the MIDL compile. For example:

```
   midl -ms_ext -cpp_cmd $(cc) -cpp_opt "-E" sample.idl
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbnetwork
KBSubcategory: NtwkRpc

Using SendMessage() As Opposed to SendDlgItemMessage()

PSS ID Number: Q12273

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

The following information describes under what circumstances it is appropriate to use either the SendMessage() or SendDlgItemMessage() function.

Both SendMessage() and SendDlgItemMessage() can be used to add strings to a list box. SendMessage() is used to send a message to a specific window using the handle to the list box. SendDlgItemMessage() is used to send a message to the child window of a given window using the list box resource ID. SendDlgItemMessage() is most often used in dialog box functions that have a handle to the dialog box and not to the child window control.

The SendDlgItemMessage() call

```
SendDlgItemMessage (hwnd, id, msg, wParam, lParam)
```

is equivalent to the following SendMessage() call:

```
hwnd2 = GetDlgItem (hwnd, id);
SendMessage (hwnd2, msg, wParam, lParam);
```

Please note that PostMessage() should never be used to talk to the child windows of dialog boxes for the following reasons:

1. PostMessage() will only return an error if the message was not posted to the control's message queue. Since many messages are sent to control return information, PostMessage() will not work, since it does not return the information to the caller.

2. 16-bit only: Messages such as the WM_SETTEXT message that include a far pointer to a string can potentially cause problems if posted using the PostMessage() function. The far pointer may point into a buffer that is inside the DS (data segment). Because PostMessage() does not process the message immediately, the DS might get moved. If the DS is moved before the message is processed, the far pointer to the buffer will be invalid.

Additional reference words: 3.00 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrMsg

Using SendMessageTimeout() in a Multithreaded Application

PSS ID Number: Q106716

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

SendMessageTimeout() is new to the Win32 application programming interface (API). The function sends a message to a window and does not return until either the window procedure processes the message or the timeout occurs.

MORE INFORMATION

This article uses the following scenario to illustrate the behavior of this function:

Suppose that Win1 and Win2 are created by the same application and that the following code is included in their window procedures:

```
   <WindowProc for window Win1>
      ...
      case <xxx>:
         ...
         SendMessageTimeout(      hWnd2,   // window handle
                            WM_USER+1,   // message to send
                               wParam,   // first message parameter
                               lParam,   // second message parameter
                           SMTO_NORMAL,   // flag *
                                  100,   // timeout (in milliseconds)
                                 &ret ); // return value
        ...
         break;

      case WM_USER+2:
        <time-consuming procedure>
         break;
```

* Note that the SMTO_NORMAL flag indicates that the calling thread can process other requests while waiting for the API to return.

```
   <WindowProc for window Win2>
      ...
      case WM_USER+1:
         ...
         SendMessage(     hWnd1,    // window handle
                     WM_USER+2,    // message to send
```

```
                        wParam,    // first message parameter
                        lParam );  // second message parameter
        OtherStuff();
        ...
        break;
```

If Win1 executes this SendMessageTimeout() and Win2 uses SendMessage() to send a message to Win1, Win1 can process the message because SMTO_NORMAL was specified. If the SendMessageTimeout() expires while the execution is currently in the window procedure for Win1, the state of the system will depend on who owns the windows.

If both windows were created by the same thread, the timeout is not used and the process proceeds exactly as if SendMessage() was being used. If the windows are owned by different threads, the results can be unpredictable, because the timeout is restarted whenever a message or some other system event is received and processed. In other words, the receipt by Win1 of WM_USER+2 causes the timeout to restart after the message is processed. If the function executed by Win2, OtherStuff(), then uses up more than 100 milliseconds without awakening the thread that created Win1, the original SendMessageTimeout() will timeout and return. The OtherStuff() function continues to completion but any value that was to be returned to Win1 will be lost. Note that the code paths will always complete.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMisc

Using Serial Communications Under Win32s

PSS ID Number: Q105759

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

Windows NT and Windows provide significantly different serial communications application programming interfaces (APIs). Win32s does not support the Win32 Communications API.

A good approach to take in writing a Win32-based application targetted for Win32s that uses serial communications is to create a pair of dynamic-link libraries (DLLs) with the same name. One DLL will use Win32 Communications APIs and be installed under Windows NT. The other DLL will use the Universal Thunk to call a 16-bit DLL that will call the Windows Communications API. This DLL will be installed under Win32s.

For more information on the Universal Thunk, see the "Win32s Programmers Guide" included with the Software Development Kit (SDK). In addition, there is a sample in MSTOOLS\WIN32S\UT\SAMPLES\UTSAMPLE.

Additional reference words: 1.00 1.10 1.20 comm
KBCategory: kbprg
KBSubcategory: W32s

Using SetClassLong Function to Subclass a Window Class

PSS ID Number: Q32519

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

It is possible to subclass an entire window class by using the SetClassLong() function. However, doing so will only subclass windows of that class created after the call to the SetClassLong() function. Windows created before the call to the SetClassLong() function are not affected.

NOTE: In Win32, SetClassLong() only affects Windows in the same address space. For example, if you subclass EDIT, only edit controls created in your application will be subclassed.

MORE INFORMATION

Calling the SetClassLong() function with the GCL_WNDPROC index changes the class function address for that window class, creating a subclass of the window class. When a subsequent window of that class is created, the new class function address is inserted into its window structure, subclassing the new window. Windows created before the call to the SetClassLong() function (in other words, before the class function address was changed) are not subclassed.

An application should not use the SetClassLong() function to subclass standard Windows controls such as edit controls or buttons. If, for example, an application were to subclass the entire "edit" class, then subsequent edit controls created by other applications would be subclassed.

An application can subclass individual standard Windows controls that it has created by calling the SetWindowLong() function.

Additional reference words: listbox scrollbar 3.00 3.10 3.50 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Using SetThreadLocale() for Language Resources

PSS ID Number: Q99392

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

Under Windows NT, each resource loading application programming interface (API) is based on the thread's locale. Each thread has a locale--usually the default system locale.

You can change the thread locale by calling SetThreadLocale(). To obtain the language resource you want, just set the thread locale to the locale you want, then call the normal resource loading API.

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrNls WintlDev

Using SndPlaySound to Play Wave Files

PSS ID Number: Q133064

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK), version 3.1
- Microsoft Win32 Software Development Kit (SDK), version 3.5

The sndPlaySound() multimedia API function is capable of playing memory images using the SND_MEMORY flag if the waveform audio exists as a .WAV file, a wave resource, or is dynamically constructed by the application.

When using the SND_MEMORY flag, the data must be allocated from global memory using the GPTR and GMEM_SHARED flags and possess both the RIFF and PCM information.

The following code demonstrates how to use the SND_MEMORY flag.

```
   {  // Sample code to demonstrate SND_MEMORY flag and memory files.

      // Sound resource bound within executable.

      if (hWaveRes = FindResource(ghInst,"TADA","WAVE"))

         {   // Resource intact; load into GLOBAL MEMORY/GMEM_SHARED memory

            if (hGlobMem = LoadResource(ghInst,hWaveRes))

               {  // Load resource into global memory and play.

                  // Play sound resource via sndPlaySound() using
                  //  SND_MEMORY flag.

                  // Application waits until sndPlaySound completes
                  //  given SND_SYNC.

                  // SND_MEMORY (first parameter is ptr to memory image
                  //  vs. filename).

                  sndPlaySound((LPSTR)LockResource(hGlobMem),
                                                SND_SYNC | SND_MEMORY);

                  FreeResource(hGlobMem);  // Required in 16-bit
                                           //  applications.


               }  // Load resource into global memory and play.

               else MessageBox(NULL,"No resource!","Multimedia Sampler!",
                               MB_ICONHAND);

            GlobalFree(hGlobMem);
```

```
        }  // Resource found.

        else MessageBox(NULL,"Lost resource!","Multimedia Sampler!",
                        MB_ICONHAND);

   }
```

Additional reference words: 3.10 3.50 kbinf win16sdk mmio mmioOpen mmioRead
KBCategory: kbmm kbcode
KBSubcategory: MMWave

Using Tabs with Initialization File APIs

PSS ID Number: Q132180

This information applies to:

- Microsoft Win32 Software Development Kit (SDK) version 4.0

SUMMARY
=======

Windows 95 does not support the use of the tab (that is, \t) character as part of the lpszString parameter of WritePrivateProfileString or WriteProfileString. In addition, GetPrivateProfileString does not return any characters in a key that occur after a tab character.

This behavior is for backward compatibility with applications that assume that comments are separated from entries by tabs.

NOTE: Windows NT does support the use of the tab character.

MORE INFORMATION
================

If a call made to either WritePrivateProfileString or WriteProfileString that contains the tab character as part of the lpszString parameter, the desired effect will not occur. Both of these functions will return TRUE (that is, successful completion); however, any data that was to follow the tab character will be missing. The newly created or modified key of the .INI file will indicate the loss of data.

When using the GetPrivateProfileString API to retrieve a string from a specified section in an .INI file, if the string is in the form:

   lpszKey=ValuePart1 <tab> ValuePart2

then GetPrivateProfileString returns ValuePart1, but does not return ValuePart2 because of the tab.

Code to Demonstrate Behavior

The following code demonstrates this loss of data after a call to WritePrivateProfileString:

```
void main()
{
     char szBuf[200];

     if ( !WritePrivateProfileString("TEST",
           "TESTKEY",
           "Test String \t more text",
           "TEST.INI")  )
     {
```

```
            FormatMessage(
                 FORMAT_MESSAGE_FROM_SYSTEM,
                 NULL,
                 GetLastError(),
                 MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
                 szBuf,
                 199,
                 NULL );

            MessageBox(NULL, szBuf,
                  "Error calling WritePrivateProfileString",
                  MB_OK);
     }
}
```

Additional reference words: 95 4.00
KBCategory: kbenv
KBSubcategory: KrMisc

Using Temporary File Can Improve Application Performance

PSS ID Number: Q103237

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

The use of temporary files can significantly increase the performance of an application.

MORE INFORMATION

================

By using CreateFile() with the FILE_ATTRIBUTE_TEMPORARY flag, you let the system know that the file is likely to be short lived. The temporary file is created as a normal file. The system needs to do a minimal amount of lazy writes to the file system to keep the disk structures (directories and so forth) consistent. This gives the appearance that the file has been written to the disk. However, unless the Memory Manager detects an inadequate supply of free pages and starts writing modified pages to the disk, the Cache Manager's Lazy Writer may never write the data pages of this file to the disk. If the system has enough memory, the pages may remain in memory for any arbitrary amount of time. Because temporary files are generally short lived, there is a good chance the system will never write the pages to the disk.

To further increase performance, your application might mark the file as FILE_FLAG_DELETE_ON_CLOSE. This indicates to the system that when the last handle of the file is closed, it will be deleted. Although the system generally purges the cache to ensure that a file being closed is updated appropriately, because a file marked with this flag won't exist after the close, the system foregoes the cache purge.

Additional reference words: 3.10 3.50

KBCategory: kbprg

KBSubcategory: BseFileio

Using Text Bullets in a Rich Edit Control

PSS ID Number: Q129859

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95 version 4.0
  - Microsoft Win32s version 1.3

The Rich Edit control contains a built-in text bullet feature that is used to add paragraph bullets to the text. To use this feature, you must send the Rich Edit control a EM_SETPARAFORMAT message. The EM_SETPARAFORMAT message takes a pointer to a PARAFORMAT structure as its lParam parameter. In the PARAFORMAT structure, it is necessary to zero out the structure and fill in the following members:

UINT cbSize - Contains the size of the structure. Use sizeof(PARAFORMAT).

DWORD dwMask - Contains the attributes to set. For bullets, use PFM_NUMBERING | PFM_OFFSET.

WORD wNumbering - Contains the value that specifies numbering options. For bullets, set this member equal to PFN_BULLET.

LONG dxOffset - Contains the value that specifies indentation of the second line and subsequent lines, relative to the starting indentation. For bullets, this value must be positive because the bullet is displayed in the area between the starting indentation and the offset. If this value is too small, the bullets are not displayed.

Additional reference words: 1.30 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl W32s

Using the C Run-Time

PSS ID Number: Q94248

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

SUMMARY
=======

This document contains the following sections:

MORE INFORMATION
================

Section 1: Three Forms of C Run-Time (CRT) Libraries Are Available
---

There are three forms of the C Run-time library provided with the Win32 SDK:

- LIBC.LIB is a statically linked library for single-threaded programs.

- LIBCMT.LIB is a statically linked library that supports multithreaded programs.

- CRTDLL.LIB is an import library for CRTDLL.DLL that also supports multithreaded programs. CRTDLL.DLL itself is part of Windows NT.

Microsoft Visual C++ 32-bit edition contains these three forms as well, however, the CRT in a DLL is named MSVCRT.LIB. The DLL is redistributable. Its name depends on the version of VC++ (ie MSVCRT10.DLL or MSVCRT20.DLL). Note however, that MSVCRT10.DLL is not supported on Win32s, while CRTDLL.LIB is supported on Win32s. MSVCRT20.DLL comes in two versions: one for Windows NT and the other for Win32s.

Section 2: Using the CRT Libraries When Building a DLL

When building a DLL which uses any of the C Run-time libraries, in order to ensure that the CRT is properly initialized, either

1. the initialization function must be named DllMain() and the entry point must be specified with the linker option -entry:_DllMainCRTStartup@12

   - or -

2. the DLL's entry point must explicitly call CRT_INIT() on process attach and process detach

This permits the C Run-time libraries to properly allocate and initialize C Run-time data when a process or thread is attaching to the DLL, to properly clean up C Run-time data when a process is detaching from the DLL, and for global C++ objects in the DLL to be properly constructed and destructed.

The Win32 SDK samples all use the first method. Use them as an example. Also refer to the Win32 Programmer's Reference for DllEntryPoint() and the Visual C++ documentation for DllMain(). Note that DllMainCRTStartup() calls CRT_INIT() and CRT_INIT() will call your application's DllMain(), if it exists.

If you wish to use the second method and call the CRT initialization code yourself, instead of using DllMainCRTStartup() and DllMain(), there are two techniques:

1. if there is no entry function which performs initialization code, simply specify CRT_INIT() as the entry point of the DLL. Assuming that you've included NTWIN32.MAK, which defines DLLENTRY as "@12", add the following option to the DLL's link line:

   ```
   -entry:_CRT_INIT$(DLLENTRY)
   ```

   - or -

2. if you *do* have your own DLL entry point, do the following in the entry point:

   a. Use this prototype for CRT_INIT():

   ```
   BOOL WINAPI _CRT_INIT(HINSTANCE hinstDLL, DWORD fdwReason,
      LPVOID lpReserved);
   ```

   For information on CRT_INIT() return values, see the documentation DllEntryPoint; the same values are returned.

   b. On DLL_PROCESS_ATTACH and DLL_THREAD_ATTACH (see "DllEntryPoint" in the Win32 API reference for more information on these flags), call CRT_INIT(), first, before any C Run-time functions are called or any floating-point operations are performed.

   c. Call your own process/thread initialization/termination code.

   d. On DLL_PROCESS_DETACH and DLL_THREAD_DETACH, call CRT_INIT() last, after all C Run-time functions have been called and all floating-point operations are completed.

   Be sure to pass on to CRT_INIT() all of the parameters of the entry point; CRT_INIT() expects those parameters, so things may not work reliably if they are omitted (in particular, fdwReason is required to determine whether process initialization or termination is needed).

Below is a skeleton sample entry point function that shows when and how to make these calls to CRT_INIT() in the DLL entry point:

```
BOOL WINAPI DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason,
    LPVOID lpReserved)
{
if (fdwReason == DLL_PROCESS_ATTACH || fdwReason == DLL_THREAD_ATTACH)
    if (!_CRT_INIT(hinstDLL, fdwReason, lpReserved))
        return(FALSE);

if (fdwReason == DLL_PROCESS_DETACH || fdwReason == DLL_THREAD_DETACH)
    if (!_CRT_INIT(hinstDLL, fdwReason, lpReserved))
        return(FALSE);
return(TRUE);
}
```

NOTE that this is *not* necessary if you are using DllMain() and -entry:_DllMainCRTStartup@12.

Section 3: Using NTWIN32.MAK to Simplify the Build Process

There are macros defined in NTWIN32.MAK that can be used to simplify your makefiles and to ensure that they are properly built to avoid conflicts. For this reason, Microsoft highly recommends using NTWIN32.MAK and the macros therein.

For compilation, use:

$(cvarsdll)	for apps/DLLs using CRT in a DLL

For linking, use one of the following:

$(conlibsdll)	for console apps/DLLs using CRT in a DLL
$(guilibsdll)	for GUI apps using CRT in a DLL

Section 4: Problems Encountered When Using Multiple CRT Libraries

If an application that makes C Run-time calls links to a DLL that also makes C Run-time calls, be aware that if they are both linked with one of the statically-linked C Run-time libraries (LIBC.LIB or LIBCMT.LIB), the .EXE and DLL will have separate copies of all C Run-time functions and global variables. This means that C Run-time data cannot be shared between the .EXE and the DLL. Some of the problems that can occur as a result are:

- Passing buffered stream handles from the .EXE/DLL to the other module
- Allocating memory with a C Run-time call in the .EXE/DLL and reallocating or freeing it in the other module
- Checking or setting the value of the global errno variable in the .EXE/DLL and expecting it to be the same in the other module. A related problem is calling perror() in the opposite module from where the C Run-time error occurred, since perror() uses errno.

To avoid these problems, link both the .EXE and DLL with CRTDLL.LIB or

MSVCRT.LIB, which allows both the .EXE and DLL to use the common set of functions and data contained within CRT in a DLL, and C Run-time data such as stream handles can then be shared by both the .EXE and DLL.

Section 5: Mixing Library Types

You can link your DLL with CRTDLL.LIB/MSVCRT.LIB regardless of what your .EXE is linked with if you avoid mixing CRT data structures and passing CRT file handles or CRT FILE* pointers to other modules.

When mixing library types adhere to the following:

- CRT file handles may only be operated on by the CRT module that created them.

- CRT FILE* pointers may only be operated on by the CRT module that created them.

- Memory allocated with the CRT function malloc() may only be freed or reallocated by the CRT module that allocated it.

To illustrate this, consider the following example:

  - .EXE is linked with MSVCRT.LIB
  - DLL A is linked with LIBCMT.LIB
  - DLL B is linked with CRTDLL.LIB

  If the .EXE creates a CRT file handle using _create() or _open(), this file handle may only be passed to _lseek(), _read(), _write(), _close(), etc. in the .EXE file. Do not pass this CRT file handle to either DLL. Do not pass a CRT file handle obtained from either DLL to the other DLL or to the .EXE.

  If DLL A allocates a block of memory with malloc(), only DLL A may call free(), _expand(), or realloc() to operate on that block. You cannot call malloc() from DLL A and try to free that block from the .EXE or from DLL B.

  NOTE: If all three modules were linked with CRTDLL.LIB or all three were linked with MSVCRT.LIb, these restrictions would not apply.

When linking DLLs with LIBC.LIB, be aware that if there is a possibility that such a DLL will be called by a multithreaded program, the DLL will not support multiple threads running in the DLL at the same time, which can cause major problems. If there is a possibility that the DLL will be called by multithreaded programs, be sure to link it with one of the libraries that support multithreaded programs (LIBCMT.LIB, CRTDLL.LIB or MSVCRT.LIB).

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsMisc

Using the Call-Attributed Profiler (CAP)

PSS ID Number: Q118890

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, versions 3.1 and 3.5

SUMMARY

The call-attributed profiler (CAP) allows you to profile function calls within an application.

MORE INFORMATION

To profile an application using CAP, perform the following steps:

1. Replace the Windows NT system DLLs that your application uses with DLLs that contain debugging information. To find out which DLLs your application uses, run the APF32CVT.EXE utility provided with the Win32 SDK. The syntax to use is as follows:

   ```
   apf32cvt <application>
   ```

   This will list the DLLs to which your program is linked. The system DLLs that contain debugging information can be found in the \SUPPORT\DEBUGDLL\i386 of the Win32 SDK CD. Rename the DLLs in your WINNT\SYSTEM32 directory and copy the debugging DLLs to the WINNT\SYSTEM32 directory. You will need to reboot the machine to use the DLLs.

2. Recompile your application. If you are using the NTWIN32.MAK file in your makefile, all you need to do is to set the environment variable PROFILE=on. Otherwise, add /Gh and /Zd to the compiler options yourself and be sure that you are linking with the options "-debugtype:coff" and "-debug:partial,mapped".

3. Place a CAP.INI file in either the root directory of the drive, the application directory, the WINDOWS directory, or the root of the C drive. The CAP.INI file specifies the applications for which the profiler will gather information. At minimum, CAP.ini must contain the following:

   ```
   [EXES]
   <app>.exe
   [PATCH IMPORTS]
   <app>.exe
   [PATCH CALLERS]
   ```

   where <app> is the application to be profiled. The file CAP.TXT included in the \BIN directory of the SDK provides an excellent

example.

4. Run the application. The profiling information is gathered
   and stored in a file with the same base name as the application
   and a .END extension. This information is in an ASCII format and
   can be viewed by any text editor. You can also use the CAPVIEW
   sample to view a graphical representation of the information.

Walter Oney points out in "Removing Bottlenecks from Your Program with Windows NT Performance-tuning Tools," from the April 1994 edition of "Microsoft Systems Journal," that the Visual C++ linker does not correctly generate debugging information that CAP can use. This is not correct. The problem is that the SDK 3.1 linker uses "-debug:mapped" by default, but the Visual C++ linker does not. Adding the switch to the link line (as in step 2, above) corrects this problem.

A common problem is for the profiling output to have "??? : ???" in place of the function names from your application. For example:

```
   1   ??? : ??? (Address=0x77889a1b)    1        4717        4717
             4717        4717        4717         n/a         n/a
```

This occurs if you use the wrong linker options. You should use "-debugtype:coff" and "-debug:partial,mapped".

Another common problem is to have function pointers instead of the Win32 API names. For example:

```
   1    0x77e9b10f                        1      1577         1577
             1577        1577        1577         n/a         n/a
```

This happens when you do not replace the system DLLs that your application calls with the DLLs that contain debugging information.

REFERENCES
==========

The release notes for CAP.TXT can be found in the MSTOOLS\BIN directory.

The best source of information is "Optimizing Windows NT" by Russ Blake in the "Windows NT Resource Kit, Vol. 3".

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsMisc

Using the DeferWindowPos Family of Functions

PSS ID Number: Q87345

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

In the Microsoft Windows graphical environment, an application can use the BeginDeferWindowPos, DeferWindowPos, and EndDeferWindowPos functions when it moves or sizes a set of windows simultaneously. Using these functions avoids unnecessary screen painting, which would occur if the windows were moved individually.

The eighth parameter to the DeferWindowPos function can be any one of eight flag values that affect the size and position of each moved or sized window. One of the flags, SWP_NOREDRAW, disables repainting and prevents Windows from displaying any changes to the screen. This flag effects both the client and nonclient areas of the window. Any portion of its parent window uncovered by the move or size operation must be explicitly invalidated and redrawn.

If the moved or sized windows are child windows or pop-up windows, then the SWP_NOREDRAW flag has the expected effect. However, if the window is an edit control, a combo box control, or a list box control, then specifying SWP_NOREDRAW has no effect; the control is drawn at its new location and its previous location is not erased. This behavior is caused by the manner in which these three control classes are painted. Buttons and static controls function normally.

To work around this limitation and move a group of edit, list box, and combo box controls in a visually pleasing manner, perform the following three steps:

1. Use the ShowWindow function to hide all of the controls.

2. Move or size the controls as required with the MoveWindow and SetWindowPos functions.

3. Use the ShowWindow function to display all of the controls.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 combobox listbox
KBCategory: kbui
KBSubcategory: UsrWndw

Using the Document Properties Dialog Box

PSS ID Number: Q118622

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

A number of applications display a printer-configuration dialog box titled "Document Properties" when the user chooses File from the application menu, chooses Print, and then chooses the Setup button.

The DocumentProperties() API is used to display this dialog box. You need to call DocumentProperties() with the DM_IN_PROMPT bit set in the last parameter (fMode). You also need to call OpenPrinter() before calling DocumentProperties().

MORE INFORMATION

The following code demonstrates how to call DocumentProperties():

Sample Code

```
   HDC         hPrnDC;
   LPDEVMODE   lpDevMode = NULL;
   LPDEVNAMES  lpDevNames;
   LPSTR       lpszDriverName;
   LPSTR       lpszDeviceName;
   LPSTR       lpszPortName;
   PRINTDLG    pd;
   HANDLE      hPrinter;
   int         nDMSize;
   HANDLE      hDevMode;
   NPDEVMODE   npDevMode;
   DEVMODE     DevModeIn;

   // Get the defaults without displaying any dialog boxes.

   pd.Flags = PD_RETURNDEFAULT;
   pd.hDevNames = NULL;
   pd.hDevMode = NULL;
   pd.lStructSize = sizeof(PRINTDLG);
   PrintDlg((LPPRINTDLG)&pd);

   lpDevNames = (LPDEVNAMES)GlobalLock(pd.hDevNames);
   lpszDriverName = (LPSTR)lpDevNames + lpDevNames->wDriverOffset;
```

```
   lpszDeviceName = (LPSTR)lpDevNames + lpDevNames->wDeviceOffset;
   lpszPortName   = (LPSTR)lpDevNames + lpDevNames->wOutputOffset;

   OpenPrinter(lpszDeviceName,&hPrinter,NULL);

   // A zero for last param returns the size of buffer needed.

   nDMSize = DocumentProperties(hWnd,hPrinter,lpszDeviceName,NULL,NULL,0);
   if ((nDMSize < 0) || !(hDevMode = LocalAlloc (LHND, nDMSize)))
      return NULL;

   npDevMode = (NPDEVMODE) LocalLock (hDevMode);

   // Fill in the rest of the structure.

   lstrcpy (DevModeIn.dmDeviceName, lpszDeviceName);
   DevModeIn.dmSpecVersion    = 0x300;
   DevModeIn.dmDriverVersion  = 0;
   DevModeIn.dmSize           = sizeof (DevModeIn);
   DevModeIn.dmDriverExtra    = 0;

   // Display the "Document Properties" dialog box.

   DocumentProperties(hWnd,hPrinter,lpszDeviceName,npDevMode,&DevModeIn,
      DM_IN_PROMPT|DM_OUT_BUFFER);

   // Get the printer DC.

   hPrnDC = CreateDC
(lpszDriverName,lpszDeviceName,lpszPortName,(LPSTR)npDevMode);
   LocalUnlock (hDevMode);

   //  Use the printer DC.

   ...
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprint
KBSubcategory: GdiPrn

Using the DRAWPATTERNRECT Escape in Windows

PSS ID Number: Q75380

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The DRAWPATTERNRECT escape is implemented by the PCL/HP driver for Hewlett-Packard (HP) LaserJet printers and compatibles. The escape is used to draw a patterned rectangle without using the graphics banding bitmap. Using this escape can enhance the performance for many applications, particularly when a majority of users have LaserJet-compatible printers.

MORE INFORMATION

The HP LaserJet Plus and all later LaserJets implement a command called a rule. A rule is a rectangle filled with some pattern, such as a black rule, a quasi-halftone gray scale, or a hatch pattern.

The output does not go through the graphics banding bitmap (it is actually sent to the printer in the text band). The DRAWPATTERNRECT escape can be used to print line and block graphics in the text band without using graphics banding at all. Because many applications use only horizontal and vertical lines or blocks, this is a significant optimization.

An application should determine support for rules using the QUERYESCSUPPORT escape. In particular, the application should not check for the PCL/HP driver, since other page printer drivers may implement the escape as well.

There are some limitations to the escape. First, rules drawn with DRAWPATTERNRECT are not subject to clipping regions in the Device Context (DC). Second, rules cannot be opaqued; no white pixel in the graphics band will erase a pixel drawn by a rule (the same limitation occurs for PCL text). Once a rule is drawn, it cannot be erased.

If these limitations are acceptable, if all graphics on the page are likely to be horizontal and vertical lines, and if a significant number of users are expected to have LaserJet-type printers (which is the case for most Windows-based applications), the DRAWPATTERNRECT escape should be used.

If for any reason DRAWPATTERNRECT cannot be used, then the application should generally use the PatBlt function. If the device is a plotter, the Rectangle function should be used. In the case of the PatBlt function, if a black rectangle is to be printed, the BLACKNESS raster operator (ROP) should be used to avoid the overhead of selecting and later deselecting a black brush into the printer DC.

Additional reference words: 3.00 3.10 3.50 4.00 95 HPPCL HP-PCL
KBCategory: kbprint
KBSubcategory: GdiPrn

Using the DS_SETFONT Dialog Box Style

PSS ID Number: Q87344

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In the Microsoft Windows graphical environment, an application can affect the appearance of a dialog box by specifying the DS_SETFONT style bit. DS_SETFONT is available only when the application creates a dialog box dynamically from a memory-resident dialog box template using the CreateDialogIndirect, CreateDialogIndirectParam, DialogBoxIndirect, or DialogBoxIndirectParam function. The second parameter to each of these functions is the handle to a global memory object that contains a DLGTEMPLATE dialog box template data structure. The dwStyle (first) member of the DLGTEMPLATE structure contains style information for the dialog box.

When an application creates a dialog box using one of these functions, Windows determines whether the template contains a FONTINFO data structure by checking for the DS_FONTSTYLE bit in the dwStyle member of the DLGTEMPLATE structure. If this bit is set, Windows creates a font for the dialog box and its controls based on the information in the FONTINFO structure. Otherwise, Windows uses the default system font to calculate the size of the dialog box and the placement and text of its controls.

If Windows creates a font based on the FONTINFO data structure, it sends a WM_SETFONT message to the dialog box. If Windows uses the system default font, it does not send a WM_SETFONT message. A dialog box can change the font of one or more of its controls by creating a font and sending a WM_SETFONT message with the font handle to the appropriate controls.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Using the FORCEFONT .HPJ Option

PSS ID Number: Q93395

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5, 3.51, and 4.0

The FORCEFONT Help project file (HPJ) option can only be used with the following font names:

```
Helv
Helvetica
Courier
Tms Rmn
Times
Symbol
```

Though Helv, Helvetica, Times, and Tms Rmn are not Windows 3.1 fonts, each of them is mapped to a 3.1 font in the [FontSubstitutes] section of WIN.INI by Windows Setup. They are mapped as follows:

```
Helv=MS Sans Serif
Tms Rmn=MS Serif
Times=Times New Roman
Helvetica=Arial
```

To force a Help file to use MS Sans Serif, the FORCEFONT option should be:

```
FORCEFONT=Helv
```

Additional reference words: 3.10 3.50 4.00 95 HC30 HC31 HCP help compiler MAPFONTSIZE
KBCategory: kbtool
KBSubcategory: TlsHlp

Using the GetWindow() Function

PSS ID Number: Q33161

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

An application can use the GetWindow function to enumerate the windows that have a specified relationship to another window. For example, an application can determine the windows that are children of the application's main window.

MORE INFORMATION

The GetWindow function returns NULL when no more windows match the specified criteria. Given a window handle, hWnd, the following code determines how many siblings the associated window has:

```
int CountSiblings(HWND hWnd)
   {
   HWND hWndNext;
   short nCount = 0;

   hWndNext = GetWindow(hWnd, GW_HWNDFIRST);
   while (hWndNext != NULL)
      {
      nCount++;
      hWndNext = GetWindow(hWndNext, GW_HWNDNEXT);
      }

   return nCount;
   }
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Using the Registry Under Win32s

PSS ID Number: Q129542

The information in this article applies to:

- Microsoft Win32s, version 1.2

SUMMARY
=======

A Win32s-based application running under Win32s is limited to the Windows version 3.1 view of the registry. This means that there are no named values. In addition, HKEY_CLASSES_ROOT is available, but not HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER.

The Windows version 3.1 registry is very limited in size. Microsoft recommends that you store only the OLE registration information in the Windows registry.

MORE INFORMATION
================

While Win32s does support some Registry APIs that Windows does not, it does not support all of the Win32 Registry APIs. For a complete list of the Registry APIs supported under Win32s, please see the file WIN32API.CSV, which is included with the Win32 Software Development Kit (SDK) and in Microsoft Visual C++ (32-bit edition).

NOTE: All supported Registry APIs, except for RegQueryValueA() and RegQueryValueExA(), return the error codes defined for Windows verion 3.1, not the codes defined by the Win32 API.

Additional reference words: 1.20
KBCategory: kbprg
KBSubcategory: W32s

Using the Windows 95 Common Controls on Windows NT and Win32s

PSS ID Number: Q125672

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY

To provide greater compatibility between Windows 95, Windows NT, and Win32s, the common control library from Windows 95 has been ported to Windows NT and Win32s starting with Windows NT version 3.51 and Win32s version 1.3.

MORE INFORMATION

The COMCTL32.DLL that provides the common controls in Windows 95 is not compatible with Windows NT or Win32s, so adding the controls to Windows NT is not as simple as copying the DLL. Also, the COMCTL32.DLL that comes with Windows NT version 3.51 and with Win32s is not redistributable. Customers that want to run programs using the new controls must be running Windows NT version 3.51 or Win32s version 1.3.

Windows NT version 3.51 and Win32s version 1.3 are targeted to be released before the release of Windows 95, so any changes in the functionality of these controls between the release of Windows NT and Win32s and the release of Windows 95 will be added to Windows NT and Win32s in their next updates.

Additional reference words: 4.00 1.30 3.51
KBCategory: kbui
KBSubcategory: UsrCtl

Using the WM_VKEYTOITEM Message Correctly

PSS ID Number: Q108941

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY

The WM_VKEYTOITEM message is sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_KEYDOWN message. The return value from this message specifies the action that the application performed in response to the message. A return value of -2 indicates that the application handled all aspects of selecting the item and requires no further action by the list box.

This is true only for certain keys such as the UP ARROW (VK_UP), DOWN ARROW (VK_DOWN), PAGE DOWN (VK_NEXT) and PAGE UP (VK_PREVIOUS) keys. All other keys are handled in the normal way by the list box, regardless of the return value from the WM_VKEYTOITEM message.

MORE INFORMATION

The documentation for the WM_VKEYTOITEM message indicates that the owner of a list box control can trap the WM_VKEYTOITEM message that is generated in response to a WM_KEYDOWN message and return -2 if the application does not want the default list box window procedure to take further action.

This is valid only for keys that are not translated into a character by the list box control in Windows. If the WM_KEYDOWN message translates to a WM_CHAR message and the application processes the WM_VKEYTOITEM message generated as a result of the keydown, the list box ignores the return value (it will go ahead and do the default processing for that character).

WM_KEYDOWN messages generated by keys such as VK_UP (UP ARROW), VK_DOWN (DOWN ARROW), VK_NEXT (PAGE DOWN) and VK_PREVIOUS (PAGE UP) are not translated to WM_CHAR messages. In such cases, trapping the WM_VKEYTOITEM message and returning a -2 prevents the list box from doing the default processing for that key.

For example, if an application traps the DOWN ARROW key and does some nondefault processing (such as moving the selection to the item two indexes below the currently selected item) and then returns -2, the list box control will not do any more processing with this message.

Alternatively, if the application trapped the "A" key, does some nondefault

processing, and returns -2, the list box code will still do the default processing. The list box will select an item in the list box that starts with an "A" (if one is present).

To trap keys that generate a char message and do special processing, the application must subclass the list box and trap both the WM_KEYDOWN and WM_CHAR messages, and process the messages appropriately in the subclass procedure.

NOTE: This discussion is for regular list boxes that are created with the LBS_WANTKEYBOARDINPUT style. If the list box is owner draw, the application must process the WM_CHARTOITEM message. For more information on the WM_CHARTOITEM message, refer to the SDK documentation.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 listbox
KBCategory: kbui
KBSubcategory: UsrCtl

Using UnregisterClass When Removing Custom Control Class

PSS ID Number: Q67248

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

In the Microsoft Windows environment, when no running application requires a custom control class, the class should be removed from memory to free the system resources it uses.

If an application registers a control class for temporary use, the application should use the UnregisterClass function when the control is no longer needed. If the application is terminated, Windows automatically removes any classes that the application registered; therefore, explicit use of UnregisterClass is not required. However, pairing calls to the RegisterClass and UnregisterClass functions is a good programming practice.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrCtl

Using volatile to Prevent Optimization of try/except

PSS ID Number: Q91149

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

The following is an example of a valid optimization that may take programmers by surprise.

1. A variable (temp) used only within the try-except body is declared outside it, and therefore is global with respect to the try.

2. Assignment to the variable (temp) is in the program only for a possible side effect of doing a read memory access through the pointer.

MORE INFORMATION

================

For example:

```
VOID
puRoutine( PULONG pu )
{
   ...
   ULONG temp;       // Just for probing
   ...
   try {
      temp = *pu;    // See if pu is a valid argument
   }

   except {
      // Handle exception
   }
}
```

The compiler optimizes and eliminates the entire try-except statement because temp is not used later.

If the value of temp were used globally, the compiler should treat the assignment to temp as volatile and do the assignment immediately even if it is overwritten later in the body of the try. The reasoning is that, at almost any point in the try body, control may jump to the except (or an exception filter). Presumably the programmer accessing the variable outside the try wants to get the current (most recently assigned) value.

The way to prevent the compiler from performing the optimization is:

```
   temp = (volatile ULONG) *pu;
```

If a temporary variable is not needed, given the example, the read access should still be specified as volatile, for example:

```
   *(volatile PULONG) pu;
```

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseExcept

Using VxDs and Software Interrupts Under Win32s

PSS ID Number: Q105760

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

Calling VxDs directly from a Win32-based application is not supported under Win32s. Win32s does not support the VxD interfaces, so the call is handled by the underlying Windows system. The Win32-based application runs with 32-bit stack and code sections, but Windows expects only 16-bit segments. Therefore, the calls to the VxD cannot be handled by Windows as expected.

To call software interrupts (such as Interrupt 2F) from a Win32-based application running under Windows 3.1 via Win32s, place the call in a 16-bit dynamic-link library (DLL) and use the Universal Thunks to access this DLL. To convert the addresses between segmented and linear address, use UTSelectorOffsetToLinear() and UTLinearToSelectorOffset().

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Using Windows Sockets Under Win32s and WOW

PSS ID Number: Q105757

The information in this article applies to:

- Microsoft Win32s versions 1.1 and 1.2

SUMMARY

=======

Win32s versions 1.1 and later provide a thunking layer for Windows Sockets. A 16-bit Windows Sockets 1.1 package must be installed on the Windows machine. Otherwise, the system will report that WINSOCK.DLL was not found. Windows NT provides a versions 1.0- and 1.1-compliant WOW (Windows on Win32) thunking layer.

MORE INFORMATION

================

There are a number of vendors that sell Windows Sockets packages. Windows Sockets support is available from Microsoft for LAN Manager version 2.2 for MS-DOS and Windows 3.1 at no additional cost. Similar support is also being shipped in "Microsoft TCP/IP for Windows For Workgroups" and the "Microsoft Network Client."

If you have LAN Manager with Microsoft TCP/IP, you can pick up everything you need from ftp.microsoft.com. Or, call Microsoft Sales at (800) 426-9400.

The following is information on how to subscribe to an Internet mailing list for Windows Socket programming as of 07/94:

Send mail to majordomo@mailbag.intel.com with a body that has

SUBSCRIBE WINSOCK <your_full_internet_email_address>.

If you want to be on the winsock hackers mailing list (for implementors of Windows sockets), use the following body in a separate piece of email.

SUBSCRIBE WINSOCK-HACKERS <your_full_internet_email_address>.

Other lists available include:

- winsnmp
- winsock-2
- ws2-app-review-board
- ws2-appletalk
- ws2-conn-oriented-media
- ws2-decnet
- ws2-generic-api-ext
- ws2-ipx-spx
- ws2-name-resolution

```
  ws2-oper-framework
  ws2-osi
  ws2-spec-clarif
  ws2-tcp-ip
  ws2-transp-review-board
  ws2-wireless

Use the command 'info <list>' to get more information about a specific
list.

Additional reference words: 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s
```

Validating User Account Passwords Under Windows NT

PSS ID Number: Q98891

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1, 3.5, and 3.51

Windows NT stores account names and passwords in the security accounts manager (SAM) database. Windows NT checks this database to validate passwords when users log on.

In Windows NT 3.1 and 3.5, there is no nonprivileged service that takes a user name and a password and returns an indication of whether or not the user account password is valid. There is a privileged service that handles this password validation; it is for use by logon processes such as winlogon.

Windows NT 3.51 introduces new Win32 APIs for logon support:

```
LogonUser
ImpersonateLoggedOnUser
CreateProcessAsUser
```

MORE INFORMATION

The SAM application programming interface (API) functions were not exposed due to their changing nature. Microsoft is working on a developer's kit that will provide guidelines and tutorial information about most of the security API functions, including the SAM APIs.

Exposing the SAM API will not compromise security because the passwords are encrypted within SAM; they are one-way encrypted such that not even SAM can decrypt them. Even a dictionary attack (encrypt an entire dictionary and see if any of the words match) would not be easy, because there is no SAM API function that will read the encrypted password.

Additional reference words: 3.10 3.50 non-privileged
KBCategory: kbprg
KBSubcategory: BseSecurity

Validating User Accounts (Impersonation)

PSS ID Number: Q96005

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1, 3.5, and 3.51

SUMMARY
=======

Some applications need the ability to execute processes in the context of another user. This impersonation restricts (or expands) the permissions of the account in which the application was executed (file access, permission to change system time, permission to shut down the system, and so forth).

For example, an administrator executes a network server program that allows remote users to log on to the system and perform actions, as if they were logged on to the system locally. Because the administrator initiated the server program and is currently logged on, all actions the server program performs will be in the security context of the administrator. If a guest user logs on remotely, he/she will have all the permissions the administrator account has.

With the Win32 API under Windows NT 3.1 and 3.5, impersonating a remote client is possible only via the ImpersonateDDEClientWindow(), ImpersonateNamedPipeClient() and RpcImpersonateClient() APIs.

Windows NT 3.51 introduces new Win32 APIs (Logon Support APIs) to deal with this problem:

   LogonUser
   ImpersonateLoggedOnUser
   CreateProcessAsUser

MORE INFORMATION
================

For versions of Windows NT prior to 3.51

A common application of impersonation is network server programs (daemons). For example, a remote login daemon needs a user to be able to to log in to a remote host and have the host impose all restrictions of the client login account.

If the daemon is using named pipes, dynamic data exchange (DDE), or a remote procedure call (RPC) (using the named pipes transport), the client account may be impersonated on the server daemon, which will impose all the restrictions of the client's user account.

Using other network interfaces (such as Windows sockets--network

programming interfaces), security cannot be monitored by the system. A workaround would be to impose password-level security on "login" to the application. The passwords would be maintained by the application in a private accounts database. However, none of the user actions are performed in the security context of the actual client user's account. Therefore, after the server/daemon has validated the client, the server must be careful to only perform actions on behalf of the client that the server knows the client should be allowed to do.

Another option is to create a network share with restricted access. The WNetAddConnection2() API can verify access to this system resources [disk or printer network resource (share)]. If the network share was set up to allow access by a restricted group of people, the WNetAddConnection2() could validate actual user accounts, maintained by Windows NT. As with the previous option, the daemon must be careful to perform only restricted actions on behalf of the client. This option could be used for file server daemons.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseSecurity

Value Returned by GetWindowLong(hWnd, GWL_STYLE)

PSS ID Number: Q83366

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

GetWindowLong(hWnd, GWL_STYLE) returns the window style information stored at the GWL_STYLE offset of the window data structure identified by hWnd. This offset contains the current state of the window, rather than the style specified when the window was created.

Windows can set and clear the following styles over the lifetime of a window: WS_CLIPSIBLINGS, WS_DISABLED, WS_HSCROLL, WS_MAXIMIZE, WS_MINIMIZE, WS_SYSMENU (for MDI child windows), WS_THICKFRAME, WS_VISIBLE, and WS_VSCROLL. Windows will not dynamically set or clear any of the other styles. An application can modify the style state at the GWL_STYLE offset at any time by calling SetWindowLong(hWnd, GWL_STYLE, dwNewLong), but Windows will not be aware that the style has changed. Windows maintains some internal flags on the window style and may use these rather than checking the GWL_STYLE offset of the window data structure.

GetWindowLong(hWnd, GWL_STYLE) returns a LONG value, which contains the currently active styles combined by the Boolean OR operator. An application can change the information stored at the GWL_STYLE offset by calling SetWindowLong(hWnd, GWL_STYLE, dwNewLong).

MORE INFORMATION
================

The following table lists the windows styles that Windows updates throughout the life span of a window. In the Change column below, an "S" indicates that the style is set and a "C" indicates that the style is cleared. The Where column lists the source module involved. Windows will change a window's styles as follows:

Style	Change	Window	Where	Why
WS_CLIPSIBLINGS				
	S	Overlapped	WMCREATE.C	Set on creation.
WS_CLIPSIBLINGS				

	S	Popup	WMCREATE.C	Set on creation.
WS_DISABLED	S & C	Any	WMACT.C	Set or cleared when EnableWindow function disables or enables window.
WS_HSCROLL	S & C	SB_HORZ	WINSBCTL.C	Set or cleared as scroll bar range changed (for window scroll bar, not the SCROLL BAR class).
WS_HSCROLL	S & C	SB_HORZ	SBRARE.C	Set or cleared in ShowScrollBar (for window scroll bar, not the SCROLL BAR class).
WS_HSCROLL	S & C	List boxes	LBOXCTL1.C	Set if scroll bar required to see contents.
WS_HSCROLL	S & C	MDI frame	MDIWIN.C	Set if required to see children.
WS_MAXIMIZE	C	Any	WMCREATE.C	On creation (reset immediately in WMMINMAX.C).
WS_MAXIMIZE	C	Any	WMMOVE.C	Cleared if window resized.
WS_MAXIMIZE	S & C	Any	WMMINMAX.C	Set if window maximized, cleared if no longer maximized.
WS_MINIMIZE	C	Any	WMCREATE.C	On creation (reset immediately in WMMINMAX.C).
WS_MINIMIZE	C	Any	WMMOVE.C	Cleared if window resized.
WS_MINIMIZE	S & C	any	WMMINMAX.C	Set if window minimized, cleared if no longer minimized.
WS_SYSMENU	S & C	MDI child	MDIMENU.C	Cleared if child is maximized and uses frame menu. Set when child no longer maximized.
WS_THICKFRAME				
	S & C	Any	WINSBCTL.C	State changed for only a few instructions during painting.

WS_VISIBLE	C	Any	WMCREATE.C	Cleared and then reset by call to ShowWindow.
WS_VISIBLE	S & C	Any	WMSHOW.C	Cleared if window hidden in ShowWindow, set if shown.
WS_VISIBLE	S & C	Any	WMSWP.C	Cleared if window hidden in ShowWindow, set if shown.
WS_VISIBLE	C	MDI client	MDIWIN.C	Cleared when current MDI child is maximized, and new child is activated. Immediately reset with call to ShowWindow.
WS_VISIBLE	S	Desktop	INLOADW.C	Ensure desktop visible.
WS_VISIBLE	S & C	Any	MSDWP.C	Set or cleared when DefWindowProc receives WM_SETREDRAW message to turn drawing on or off, respectively.
WS_VISIBLE	S & C	MDI client	MDIWIN.C	Cleared and then immediately reset to optimize painting.
WS_VSCROLL	S & C	SB_VERT	WINSBCTL.C	Set or cleared as scroll bar range changed (for window scroll bar, not the SCROLL BAR class).
WS_VSCROLL	S & C	List boxes	LBOXCTL1.C	Set if scroll bar required to see entire contents.
WS_VSCROLL	S & C	MDI frame	MDIWIN.C	Set if required to see children.
WS_VSCROLL	S & C	SB_VERT	SBRARE.C	Set/cleared in ShowScrollBar (for window scroll bar, not the SCROLL BAR class).

In addition to the information above, the GetWindowLong function always reports some style bits to be clear, as follows:

- Combo boxes always report the following styles as clear:

  CBS_HASSTRINGS, CBS_SORT, WS_BORDER, WS_HSCROLL, and WS_VSCROLL

- All edit controls report the WS_BORDER style clear.

- Multiline edit controls report the WS_HSCROLL style clear if the

control contains centered or right-justified text.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Various Ways to Access Submenus and Menu Items

PSS ID Number: Q71454

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

In calls to Microsoft Windows functions that create, modify, and destroy menus, an application can access an individual menu item by either its position or its item ID. A pop-up menu must be accessed by its position because it does not have a menu-item ID.

Specifically, when an application calls the EnableMenuItem() function to enable, disable, or dim (gray) an individual menu item, the application can specify either the MF_BYPOSITION or the MF_BYCOMMAND flag in the wEnable parameter. When the application calls EnableMenuItem() to access a pop-up menu, it must specify the MF_BYPOSITION flag.

The information below provides examples of the following:

- Retrieving a menu handle for a submenu
- Accessing a submenu
- Accessing a menu item

MORE INFORMATION

The following resource-file menu template provides the basis for the source code examples in this article. The template describes a top-level menu with two pop-up submenus. One of the submenus contains a third, nested submenu.

```
GenericMenu MENU
BEGIN
   POPUP "&Help"
   BEGIN
      MENUITEM "&About Generic...", IDM_ABOUT
   END

   POPUP "&Test"
   BEGIN
      POPUP "&Nested"
      BEGIN
         MENUITEM "&1 Beep", IDM_1BEEP
         MENUITEM "&2 Beeps", IDM_2BEEPS
      END
```

```
      END
   END
```

Retrieving the Handle to a Submenu

Code such as the following can be used to obtain handles to the menus:

```
   HMENU hMainMenu, hHelpPopup, hTestPopup, hNestedPopup;

   <other program lines>

   hMainMenu = GetMenu(hWnd);
   hHelpPopup = GetSubMenu(hMainMenu, 0);
   hTestPopup = GetSubMenu(hMainMenu, 1);
   hNestedPopup = GetSubMenu(hTestPopup, 0);
```

The second parameter of the GetSubMenu() function, nPos, is the position of the desired submenu. Positions are numbered starting at zero for the first menu item.

Disabling a Submenu

The following call disables and dims the nested pop-up menu:

```
   EnableMenuItem (hTestPopup, 0, MF_BYPOSITION | MF_GRAYED);
```

The following call disables and dims the Test pop-up menu:

```
   EnableMenuItem (hMainMenu, 1, MF_BYPOSITION | MF_GRAYED);
```

The second parameter of the EnableMenuItem() function, wIDEnabledItem, is the position of the submenu. As above, positions are numbered starting at zero. Note that the call must specify the MF_BYPOSITION flag because a pop-up menu does not have a menu-item ID.

Disabling a Menu Item

The 1 Beep menu item can be disabled and dimmed by using any one of the following calls:

```
   EnableMenuItem(hMainMenu, IDM_1BEEP, MF_BYCOMMAND | MF_GRAYED);
   EnableMenuItem(hTestPopup, IDM_1BEEP, MF_BYCOMMAND | MF_GRAYED);
   EnableMenuItem(hNestedPopup, IDM_1BEEP, MF_BYCOMMAND | MF_GRAYED);
   EnableMenuItem(hNestedPopup, 0, MF_BYPOSITION | MF_GRAYED);
```

A menu item can be specified by either by its menu-item ID value (using the MF_BYCOMMAND flag) or by its position (using the MF_BYPOSITION) flag. If the application specifies the menu-item ID value, Windows must walk the menu structure and search for a menu item with the correct ID. This implies

the each menu-item ID value must be unique for a given menu.

Other Windows Menu Functions

Although the EnableMenuItem() function is used in the example above, the same general approach is used for all Windows menu functions; access pop-up menus by position, and access menu items by position or menu-item ID.

Additional reference words: 3.00 3.10 3.50 4.00 95 dimmed unavailable
KBCategory: kbui
KBSubcategory: UsrMen

Video for Windows 1.1--Automatic Window Shift Problem

PSS ID Number: Q118390

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 SDK, version 4.0

When a video playback window is created using MCI, the Video for Windows (VfW) system software may shift the window a small distance vertically or horizontally from the original location.

The video software does this to align the window on even-byte boundaries in video memory for optimal playback speed. When the window is located on a byte boundary, the video software is not required to perform bit-intensive calculations during video playback, increasing the playback speed.

This behavior is particular to Video for Windows and cannot be controlled by the application being affected.

Additional reference words: 3.10 4.00 95
KBCategory: kbmm kbprg
KBSubcategory: MMVideo

Viewing Globals Out of Context in WinDbg

PSS ID Number: Q105583

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

When viewing global variables (either with the ? command or via the Watch window) and the variables go out of context, their values become:

```
CXX0017 Error: symbol not found
```

An example of this is when a common dialog box is open in the application. If you break into an application that is inside COMDLG32.DLL and try to do a ?gVar, where gVar is a global variable in the application, WinDbg will not find the symbol because the context is wrong. To view the value of gVar in MYAPP, use the following:

```
?{,,myapp}gVar
```

WinDbg will then have no trouble locating the symbolic information.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsWindbg

Virtual Memory and Win32s

PSS ID Number: Q124137

The information in this article applies to:

 - Microsoft Win32s versions 1.1, 1.15, 1.15a, and 1.2

SUMMARY
=======

This article discusses virtual memory issues under Win32s. These issues include how memory is managed, file mappings, and VirtualAlloc().

MORE INFORMATION
================

Memory Management

Win32s does not manage the virtual memory by itself. Win32s sits on top of the Win386 virtual memory manager (VMM). The Win32s VxD and Win386 must cooperatively manage the memory.

Windows sets the amount of virtual memory (VM) at boot time, based on the amount of free physical RAM. Windows is also responsible for managing the physical pages.

The pages that map to the Win32-based application's code and data are reserved at application initialization time. This decreases the available virtual memory but not necessarily the available RAM. In 16-bit Windows-based applications, the selectors are initialized at initialization time, but they are marked as discarded and are loaded only when the segments are touched. This may lead you to think that a Win32 version of you application takes significantly more memory than its 16-bit Windows version. However, in reality, the available memory drops more when the Win32-based application is loaded, even though the actual RAM usage during execution may be lower.

NOTE: Code pages are never backed by the .EXE file, but are always backed by the pagefile.

File Mappings

Because of the way memory is managed, you cannot have a file mapping that is larger than the amount of virtual memory available on Win32s. This works fine on Windows NT and Windows 95. Win32s allocates regular virtual memory for the memory mapped section even though it does not need swap space, and the amount of VM set by Windows is too small to use for mapping large files.

VirtualAlloc()

VirtualAlloc() reserves or commits pages in virtual memory. When virtual memory is allocated (committed), the page is still not present. Touching the page will make it present and initialized to zero. Unlike Windows NT, some of the address space on Win32s is not allocated by VirtualAlloc(). You can use VirtualQuery() for every address in the USER address space, as reported by GetSystemInfo().

VirtualAlloc() allocates private memory, so one process cannot commit or free memory reserved by another process. This is true for all Win32 platforms. However, on Win32s, the memory is still accessible by all processes because there is a shared address space provided by Windows.

NOTE: A "not enough memory" error occurs if you reserve memory using VirtualAlloc() in one application and then try to commit the memory from a second application. This happens because the call to commit from the second application is interpreted as a call to reserve. You need to commit at an address that is not available such as an address already reserved by the first application.

Additional reference words: 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

VirtualLock() Only Locks Pages into Working Set

PSS ID Number: Q94996

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT versions 3.1 and 3.5

VirtualLock() causes pages to be locked into an application's working set (virtual memory); it does not lock them absolutely into physical memory. VirtualLock() essentially means "this page is always part of the process's working set."

The system is free to swap out any virtually locked pages if it swaps out the whole process. And when the system swaps the process back in, the virtually locked pages (similar to any virtual pages) may end up residing in different real pages.

It is wise to use VirtualLock() very sparingly because it reduces the flexibility of the system.

Depending upon memory demands on the system, the memory manager may vary the number of pages a process can lock. Under typical conditions you can expect to be able to VirtualLock() approximately 28 to 32 pages.

In Windows NT 3.5, you can use SetProcessWorkingSetSize() to increase the size of the working set, and therefore increase the number of pages that VirtualLock() can lock.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMm

Watching Local Variables That Are Also Globally Declared

PSS ID Number: Q98288

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5. 3.51, and 4.0

SUMMARY
=======

Consider debugging the following program in WinDbg:

```
int x = 1;
int y = 2;

void main()
{
   int x = 2;
   x++;
   y++;
}
```

Notice that there is a global variable x and a local variable x.

Before you step into main, if you set watchpoints on x and y, the Watch window will display a value for y but for x will say "Expression cannot be evaluated." To see the value for x, use ::x and x will evaluate to the local x in main once you've stepped into main.

MORE INFORMATION
================

When debugging an application, the X86 C++ evaluator is loaded. Given this, you can use the scope resolution operator in a watch statement to view a hidden global variable. Without the use of the scope resolution operator, there is no way (short of watching it in a memory window) to watch a hidden global variable.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsWindbg

When to Select and Realize OpenGL Palettes

PSS ID Number: Q151489

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT, versions 3.51, 4.0
    - Microsoft Windows 95, version 4.0

SUMMARY

An OpenGL application must select and realize its palette before setting the current rendering context with wglMakeCurrent.

MORE INFORMATION

An OpenGL application typically makes its rendering context current one time only when the application is created, or repeatedly just prior to rendering, as in WM_PAINT for example.

In the first case, the palette must be created, selected, and realized prior to the initial wglMakeCurrent.

In the second case, the palette should be selected and realized before every call to wglMakeCurrent.

Additional reference words: 3.51 4.00 render colors hrc RGBA
KBCategory: kbgraphic
KBSubcategory: GdiMisc

When to Use Synchronous Socket Handles & Blocking Hooks

PSS ID Number: Q131623

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)
  versions 3.1, 3.5, 3.51, and 4.0

SUMMARY
=======

By default, all socket handles are opened as overlapped handles so that asynchronous I/O can be performed on them. However, in many situations you may find it preferable to have nonoverlapped (synchronous) socket handles.

For example, only nonoverlapped handles can be used with the C run-time libraries or used as standard I/O handles for a process. Under Windows NT and Windows 95, the SO_OPENTYPE socket option allows an application to open non-overlapped socket handles.

MORE INFORMATION
================

There are some Windows Sockets features that you cannot use with synchronous sockets. Here is an extract from the Winsock Help file:

> The WSAAsyncSelect call cannot be used with synchronous sockets and will fail with the error WSAEINVAL. It is also not possible to set the SO_SNDTIMEO and SO_RCVTIMEO socket options on synchronous sockets; setsockopt with these options on synchronous sockets fails with WSAEINVAL as well.

Due to the non-preemptive nature of Windows version 3.1 and Windows for Workgroups version 3.11, the Winsock specification details a mechanism by which a Winsock application can "yield" processor time. For more information, please search for WSASetBlockingHook() in the Winsock Help file.

NOTE: Use of a blocking hook is not recommended on a 32-bit platform. If a 32-bit application chooses to install a blocking hook, the blocking hook will be disabled if the application is run under Windows NT, but it will remain enabled if the application is run under Windows 95.

REFERENCES
==========

Online winsock help file

Additional reference words: 3.10 3.50 3.51 4.00
KBCategory: kbnetwork
KBSubcategory: NtwkWinsock

Where to Get Support for MS-DOS-Based Sockets Applications

PSS ID Number: Q138964

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, 4.0

SUMMARY

It is not possible to run 16-bit MS-DOS-based sockets applications under Windows NT or Windows 95. This is because the protocol stacks under Windows NT and Windows 95 use DLLs and do not expose an interrupt interface.

MORE INFORMATION

Please note that 16-bit MS-DOS-based applications are different from 32-bit console applications. It is possible to write sockets applications that make use of the winsock interface exposed by the protocol stacks under Windows NT and Windows 95. To do this, please refer to the Win32 SDK Winsock reference.

If you want to write 16-bit MS-DOS-based sockets application that run under Windows NT or Windows 95, contact the following company. They supply software (Virtual sockets library) that make it possible:

JSB Corporation
108 Whispering Pines Drive
Suite 115
Scotts Valley
California 95066
USA

Tel : (408) 438-8300
(800) 359-3408

Fax : (408) 438-8360

http://www.jsbus.com

The JSB Corporation is a vendor independent of Microsoft; we make no warranty, implied or otherwise, regarding the performance or reliability of JSB Corporation products.

Additional reference words: 3.10 3.50 3.51 4.00
KBCategory: kb3rdparty kbnetwork
KBSubcategory: NtwkWinsock

Where to Get the Microsoft SNMP Headers and Libraries

PSS ID Number: Q127902

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.5, 3.51, and 4.0
- Microsoft Windows for Workgroups SDK, version 3.11

The 32-bit SNMP headers and libraries are available in both the Microsoft Win32 SDK and Microsoft Visual C++ version 2.0 and later. These files are for use with the Microsoft implementation of SNMP. They will not work with other implementations because Microsoft does not conform to the WINSNMP standard of management APIs.

The headers and libraries for the SNMP agent (parts of SNMP.H and SNMP.LIB) can be used with Windows NT and Windows 95. The headers and libraries for the Management APIs (parts of SNMP.H, MGMTAPI.H, parts of SNMP.LIB, and MGMTAPI.LIB) are for use with Windows NT only.

Microsoft offers no 16-bit SNMP for Windows for Workgroups, however, there are other companies that do offer SNMP for Windows for Workgroups. Here are two companies that we know of that offer 16-bit SNMP:

Company: NetManage
10725 N. De Anza Blvd.
Cupertino, CA 95014
Product: Chameleon Utilities
Phone: (408) 973-7171
Fax: (408) 257-6405

FTP Software
Corporate Headquarters
2 High Street
North Andover, MA 01845-2620
Phone: (508) 685-4000
Fax: (508) 794-4488

The third-party products discussed here are manufactured by vendors independent of Microsoft; we make no warranty, implied or otherwise, regarding these products' performance or reliability.

Additional reference words: 3.50 4.00 95
KBCategory: kbnetwork kb3rdparty
KBSubcategory: NtwkSnmp

Which Windows NT (Server or Workstation) Is Running?

PSS ID Number: Q124305

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

You can determine which variant of Windows NT (Windows NT Server or Windows NT Workstation) is running by using the technique described in this article. Then you can use this information to execute code based on which variant is running.

MORE INFORMATION
================

To find out which product is currently running, you need to determine the value of the following registry entry:

HKEY_LOCAL_MACHINE\SYSTEM\
CurrentControlSet
Control\
ProductOptions

Use the following table to determine which product is running:

ProductType	Product
WINNT	Windows NT Workstation is running
SERVERNT	Windows NT Server is running
LANMANNT	Windows NT Advanced Server is running (Server as Domain controller)

The sample code creates a WhichNTProduct() function to indicate whether Windows NT Server or Windows NT Workstation in currently running. The following table gives the meaning for each return value:

Return Value	Meaning
RTN_SERVER	Windows NT Server is running
RTN_WORKSTATION	Windows NT Workstation is running
RTN_NTAS	Windows NT Advanced Server is running
RTN_UNKNOWN	Unknown product type was encountered
RTN_ERROR	Error occurred

To get extended error information, call GetLastError(). Some error checking is omitted, for brevity.

Sample Code

```
#define RTN_UNKNOWN 0
#define RTN_SERVER 1
#define RTN_WORKSTATION 2
#define RTN_NTAS 3
#define RTN_ERROR 13

unsigned int WhichNTProduct(void)

DWORD
WhichNTProduct(
    void
    )
{
    #define MY_BUFSIZE 32 // arbitrary. Use dynamic allocation
    HKEY hKey;
    TCHAR szProductType[MY_BUFSIZE];
    DWORD dwBufLen=MY_BUFSIZE;
    LONG lRet;

    if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,

TEXT("SYSTEM\\CurrentControlSet\\Control\\ProductOptions"),
                    0,
                    KEY_QUERY_VALUE,
                    &hKey) != ERROR_SUCCESS) return RTN_ERROR;

    lRet = RegQueryValueEx(hKey,
                    TEXT("ProductType"),
                    NULL,
                    NULL,
                    (LPBYTE)szProductType,
                    &dwBufLen);

    RegCloseKey(hKey);

    if(lRet != ERROR_SUCCESS) return RTN_ERROR;

    // check product options, in order of likelihood
    if(lstrcmpi(TEXT("WINNT"), szProductType) == 0) return RTN_WORKSTATION;
    if(lstrcmpi(TEXT("SERVERNT"), szProductType) == 0) return RTN_SERVER;
    if(lstrcmpi(TEXT("LANMANNT"), szProductType) == 0) return RTN_NTAS;

    // else return Unknown
    return RTN_UNKNOWN;
}
```

Additional reference words: 3.10 3.50
KBCategory: kbprg kbcode
KBSubcategory: BseMisc

Why LoadLibraryEx() Returns an HINSTANCE

PSS ID Number: Q102128

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

In the Win32 Help files, LoadLibrary() is typed to return a HANDLE, while LoadLibraryEx() is prototyped to return a HINSTANCE.

An HINSTANCE return from LoadLibraryEx() is useful because processes that load dynamic-link libraries (DLLs) do not necessarily want the overhead of having to page in code for a DllEntryPoint routine when the DLL does not need to initialize information. This is especially useful when you have multiple threads that attach to already loaded DLLs. In this case, you may want to not implicitly load via LoadLibrary() and instead use LoadLibraryEx() to explicitly load without having to page in the code for every attach.

LoadLibraryEx() is also useful if you want to retrieve resources from a DLL or an EXE. In this case, you would use LoadLibraryEx() to load the module you want into your address space, without executing DLLEntryPoint, and then use the resource application programming interfaces (APIs) to access the data.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseDll

Win32 Drag and Drop Server

PSS ID Number: Q105530

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The supported method for creating a drag and drop server is to use OLE (Object Linking and Embedding) version 2.0. This works on Windows, Windows NT, and Windows 95 and will work on future versions of these operating systems.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDnd

Win32 Equivalents for C Run-Time Functions

PSS ID Number: Q99456

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

=======

Many of the C Run-time functions have direct equivalents in the Win32 application programming interface (API). This article lists the C Run-time functions by category with their Win32 equivalents or the word "none" if no equivalent exists.

MORE INFORMATION

================

NOTE: the functions that are followed by an asterisk (*) are part of the 16-bit C Run-time only. Functions that are unique to the 32-bit C Run-time are listed separately in the last section. All other functions are common to both C Run-times.

Buffer Manipulation

_memccpy	none
memchr	none
memcmp	none
memcpy	CopyMemory
_memicmp	none
memmove	MoveMemory
memset	FillMemory, ZeroMemory
_swab	none

Character Classification

---

isalnum	IsCharAlphaNumeric
isalpha	IsCharAlpha, GetStringTypeW (Unicode)
__isascii	none
iscntrl	none, GetStringTypeW (Unicode)
__iscsym	none
__iscsymf	none
isdigit	none, GetStringTypeW (Unicode)
isgraph	none
islower	IsCharLower, GetStringTypeW (Unicode)
isprint	none
ispunct	none, GetStringTypeW (Unicode)
isspace	none, GetStringTypeW (Unicode)
isupper	IsCharUpper, GetStringTypeW (Unicode)
isxdigit	none, GetStringTypeW (Unicode)

__toascii	none
tolower	CharLower
_tolower	none
toupper	CharUpper
_toupper	none

Directory Control

_chdir	SetCurrentDirectory
_chdrive	SetCurrentDirectory
_getcwd	GetCurrentDirectory
_getdrive	GetCurrentDirectory
_mkdir	CreateDirectory
_rmdir	RemoveDirectory
_searchenv	SearchPath

File Handling

_access	none
_chmod	SetFileAttributes
_chsize	SetEndOfFile
_filelength	GetFileSize
_fstat	See Note 5
_fullpath	GetFullPathName
_get_osfhandle	none
_isatty	GetFileType
_locking	LockFileEx
_makepath	none
_mktemp	GetTempFileName
_open_osfhandle	none
remove	DeleteFile
rename	MoveFile
_setmode	none
_splitpath	none
_stat	none
_umask	none
_unlink	DeleteFile

Creating Text Output Routines

_displaycursor*	SetConsoleCursorInfo
_gettextcolor*	GetConsoleScreenBufferInfo
_gettextcursor*	GetConsoleCursorInfo
_gettextposition*	GetConsoleScreenBufferInfo
_gettextwindow*	GetConsoleWindowInfo
_outtext*	WriteConsole
_scrolltextwindow*	ScrollConsoleScreenBuffer
_settextcolor*	SetConsoleTextAttribute
_settextcursor*	SetConsoleCursorInfo
_settextposition*	SetConsoleCursorPosition
_settextwindow*	SetConsoleWindowInfo
_wrapon*	SetConsoleMode

Stream Routines

```
    clearerr                 none
    fclose                   CloseHandle
   _fcloseall                none
   _fdopen                   none
    feof                     none
    ferror                   none
    fflush                   FlushFileBuffers
    fgetc                    none
   _fgetchar                 none
    fgetpos                  none
    fgets                    none
   _fileno                   none
   _flushall                 none
    fopen                    CreateFile
    fprintf                  none
    fputc                    none
   _fputchar                 none
    fputs                    none
    fread                    ReadFile
    freopen (std handles)    SetStdHandle
    fscanf                   none
    fseek                    SetFilePointer
    fsetpos                  SetFilePointer
   _fsopen                   CreateFile
    ftell                    SetFilePointer (check return value)
    fwrite                   WriteFile
    getc                     none
    getchar                  none
    gets                     none
   _getw                     none
    printf                   none
    putc                     none
    putchar                  none
    puts                     none
   _putw                     none
    rewind                   SetFilePointer
   _rmtmp                    none
    scanf                    none
    setbuf                   none
    setvbuf                  none
   _snprintf                 none
    sprintf                  wsprintf
    sscanf                   none
   _tempnam                  GetTempFileName
    tmpfile                  none
    tmpnam                   GetTempFileName
    ungetc                   none
    vfprintf                 none
    vprintf                  none
   _vsnprintf                none
    vsprintf                 wvsprintf

Low-Level I/O
-------------
   _close                   _lclose, CloseHandle
```

_commit	FlushFileBuffers
_creat	_lcreat, CreateFile
_dup	DuplicateHandle
_dup2	none
_eof	none
_lseek	_llseek, SetFilePointer
_open	_lopen, CreateFile
_read	_lread, ReadFile
_sopen	CreateFile
_tell	SetFilePointer (check return value)
_write	_lread

Console and Port I/O Routines

_cgets	none
_cprintf	none
_cputs	none
_cscanf	none
_getch	ReadConsoleInput
_getche	ReadConsoleInput
_inp	none
_inpw	none
_kbhit	PeekConsoleInput
_outp	none
_outpw	none
_putch	WriteConsoleInput
_ungetch	none

Memory Allocation

_alloca	none
_bfreeseg*	none
_bheapseg*	none
calloc	GlobalAlloc
_expand	none
free	GlobalFree
_freect*	GlobalMemoryStatus
_halloc*	GlobalAlloc
_heapadd	none
_heapchk	none
_heapmin	none
_heapset	none
_heapwalk	none
_hfree*	GlobalFree
malloc	GlobalAlloc
_memavl	GlobalMemoryStatus
_memmax	GlobalMemoryStatus
_msize*	GlobalSize
realloc	GlobalReAlloc
_set_new_handler	none
_set_hnew_handler*	none
_stackavail*	none

Process and Environment Control Routines

abort	none
assert	none
atexit	none
_cexit	none
_c_exit	none
_exec functions	none
exit	ExitProcess
_exit	ExitProcess
getenv	GetEnvironmentVariable
_getpid	GetCurrentProcessId
longjmp	none
_onexit	none
perror	FormatMessage
_putenv	SetEnvironmentVariable
raise	RaiseException
setjmp	none
signal (ctrl-c only)	SetConsoleCtrlHandler
_spawn functions	CreateProcess
system	CreateProcess

String Manipulation

strcat, wcscat	lstrcat
strchr, wcschr	none
strcmp, wcscmp	lstrcmp
strcpy, wcscpy	lstrcpy
strcspn, wcscspn	none
_strdup, _wcsdup	none
strerror	FormatMessage
_strerror	FormatMessage
_stricmp, _wcsicmp	lstrcmpi
strlen, wcslen	lstrlen
_strlwr, _wcslwr	CharLower, CharLowerBuffer
strncat, wcsncat	none
strncmp, wcsncmp	none
strncpy, wcsncpy	none
_strnicmp, _wcsnicmp	none
_strnset, _wcsnset	FillMemory, ZeroMemory
strpbrk, wcspbrk	none
strrchr, wcsrchr	none
_strrev, _wcsrev	none
_strset, _wcsset	FillMemory, ZeroMemory
strspn, wcsspn	none
strstr, wcsstr	none
strtok, wcstok	none
_strupr, _wcsupr	CharUpper, CharUpperBuffer

MS-DOS Interface

_bdos*	none
_chain_intr*	none
_disable*	none
_dos_allocmem*	GlobalAlloc
_dos_close*	CloseHandle
_dos_commit*	FlushFileBuffers

_dos_creat*	CreateFile
_dos_creatnew*	CreateFile
_dos_findfirst*	FindFirstFile
_dos_findnext*	FindNextFile
_dos_freemem*	GlobalFree
_dos_getdate*	GetSystemTime
_dos_getdiskfree*	GetDiskFreeSpace
_dos_getdrive*	GetCurrentDirectory
_dos_getfileattr*	GetFileAttributes
_dos_getftime*	GetFileTime
_dos_gettime*	GetSystemTime
_dos_getvect*	none
_dos_keep*	none
_dos_open*	OpenFile
_dos_read*	ReadFile
_dos_setblock*	GlobalReAlloc
_dos_setdate*	SetSystemTime
_dos_setdrive*	SetCurrentDirectory
_dos_setfileattr*	SetFileAttributes
_dos_setftime*	SetFileTime
_dos_settime*	SetSystemTime
_dos_setvect*	none
_dos_write*	WriteFile
_dosexterr*	GetLastError
_enable*	none
_FP_OFF*	none
_FP_SEG*	none
_harderr*	See Note 1
_hardresume*	See Note 1
_hardretn*	See Note 1
_int86*	none
_int86x*	none
_intdos*	none
_intdosx*	none
_segread*	none

Time
----

asctime	See Note 2
clock	See Note 2
ctime	See Note 2
difftime	See Note 2
_ftime	See Note 2
_getsystime	GetLocalTime
gmtime	See Note 2
localtime	See Note 2
mktime	See Note 2
_strdate	See Note 2
_strtime	See Note 2
time	See Note 2
_tzset	See Note 2
_utime	SetFileTime

Virtual Memory Allocation

_vfree*	See Note 3
_vheapinit*	See Note 3
_vheapterm*	See Note 3
_vload*	See Note 3
_vlock*	See Note 3
_vlockcnt*	See Note 3
_vmalloc*	See Note 3
_vmsize*	See Note 3
_vrealloc*	See Note 3
_vunlock*	See Note 3

32-Bit C Run Time

_beginthread	CreateThread
_cwait	WaitForSingleObject w/ GetExitCodeProcess
_endthread	ExitThread
_findclose	FindClose
_findfirst	FindFirstFile
_findnext	FindNextFile
_futime	SetFileTime
_get_osfhandle	none
_open_osfhandle	none
_pclose	See Note 4
_pipe	CreatePipe
_popen	See Note 4

NOTE 1: The _harderr functions do not exist in the Win32 API. However, much of their functionality is available through structured exception handling.

NOTE 2: The time functions are based on a format that is not used in Win32. There are specific Win32 time functions that are documented in the Help file.

NOTE 3: The virtual memory functions listed in this document are specific to the MS-DOS environment and were written to access memory beyond the 640K of RAM available in MS-DOS. Because this limitation does not exist in Win32, the standard memory allocation functions should be used.

NOTE 4: While _pclose() and _popen() do not have direct Win32 equivalents, you can (with some work) simulate them with the following calls:

_popen	CreatePipe
	CreateProcess
_pclose	WaitForSingleObject
	CloseHandle

NOTE 5: GetFileInformationByHandle() is the Win32 equivalent for the _fstat() C Run-time function. However, GetFileInformationByHandle() is not supported by Win32s version 1.1. It is supported in Win32s 1.2. GetFileSize(), GetFileAttributes(), GetFileTime(), and GetFileTitle() are supported by Win32s 1.1 and 1.2.

Additional reference words: 3.10 3.50 4.00 95

KBCategory: kbprg
KBSubcategory: BseMisc

Win32 Graphical Setup Over Network Drives

PSS ID Number: Q98838

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.1 and 3.5

The graphical setup for the Win32 Software Development Kit (SDK) does not include network drives in the list of drives to choose from, however, the setup program does support installing to network drives.

In addition, you can use the manual install program, manual.bat, to install to local or network drives.

Additional reference words: 3.10 3.50
KBCategory: kbsetup
KBSubcategory: Setins

Win32 Priority Class Mechanism and the START Command

PSS ID Number: Q90910

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

The Win32 priority class mechanism is exposed through CMD.EXE's START command.

START accepts the following switches:

/LOW	-	Start the command in the idle priority class.
/NORMAL	-	Start the command in the normal priority class (this is the default).
/HIGH	-	Start the command in the high priority class.
/REALTIME	-	Start the command in the real-time priority class.

For a complete list of START switches, type the following command at the Windows NT command prompt:

```
start /?
```

Win32 has also been modified to inherit priority class if the parent's priority class is idle; thus, a command such as

```
start /LOW nmake
```

causes build and all descendants (compiles, links, and so on) to run in the idle priority class. Use this method to do a real background build that will not interfere with anything else on your system.

A command such as

```
start /HIGH nmake
```

runs BUILD.EXE in the high priority class, but all descendants run in the normal priority class.

MORE INFORMATION

Be very careful with START /HIGH and START /REALTIME. If you use either of these switches to start applications that require a lot of cycles, the applications will get all the cycles they ask for, which may cause the

system to appear hung.

Additional reference words: 3.10
KBCategory: kbprg
KBSubcategory: BseProcThrd

Win32 SDK Knowledge Base Available as Help File (JUN 1995)

PSS ID Number: Q106544

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) versions 3.1, 3.5, 3.51, and 4.0
- Microsoft Win32s, versions 1.2 and 1.25a

The Win32 SDK Knowledge Base has been compiled into a help file.

ARCHIVE NAME	FILENAME	TOPIC	MODIFY DATE
WIN32KB.EXE	WIN32KB.HLP	Win32 SDK	28-JUN-1995

Download WIN32KB.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download WIN32KB.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \SOFTLIB\MSLFILES directory
  Get WIN32KB.EXE

Additional reference words: 1.20 3.10 3.50 4.00 95
KBCategory: kbref kbfile
KBSubcategory: GenSDK

Win32 Shell Dynamic Data Exchange (DDE) Interface

PSS ID Number: Q105446

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51

SUMMARY

=======

Information on the DDE Interface to Program Manager can be found in the Win32 application programming interface (API) reference under the topic "Shell Dynamic Data Exchange Interface."

The SDK contains a sample program that interfaces with Program Manager. The sample can be found in MSTOOLS\SAMPLES\DDEML\DDEPROG.

MORE INFORMATION

================

AppProperties cannot be used to get the item icon, description, or working directory, as it can in Windows 3.1. Therefore, GetIcon(), GetDescription(), and GetWorkingDir() do not work in Windows NT. However, AppProperties can still be used to dump out the contents of a group, by specifying the group name in lParam.

Here's how a Win32-based application can get the item icon, the description, and the working directory:

1. Initiate a conversation with the Shell as follows

```
SendMessage( -1, WM_DDE_INITIATE, hWndApp, lParam );
```

   where lParam points to an atom representing:

```
LOWORD | HIWORD
-----------------------------------------------------------

Shell  | AppIcon        : To get an item's icon
Shell  | AppDescription : To get an item's description
Shell  | AppWorkingDir  : To get an item's working directory
```

2. Get the item DDE number.

   The DDE number is stored by Program Manager in the STARUPINFO structure of the application when the application is started. The application can get the startup information with:

```
GetStartupInfo( &StartupInfo );
```

   The field lpReserved in the STARUPINFO structure is in the

following format

```
dde.#, hotkey.##
```

where the DDE number is # and the hot key for the item is ##.

3. Request data as follows

```
SendMessage( hwndProgMan, WM_DDE_REQUEST, hwndApp, lParam );
```

where the lParam HIWORD is the item's DDE number obtained in step 2.

4. The data is returned in lParam of WM_DDE_DATA message. The DDE data value is a string for AppDescription and AppWorkingDir DDE transactions. For AppIcon, the data value has the following structure:

```
typedef struct _PMIconData {
   DWORD dwResSize;
   DWORD dwVer;
   BYTE iResource;  // icon resource
} PMICONDATA, *LPPMICONDATA;
```

To create the icon, the application must call:

```
hIcon = CreateIconFromResource((LPBYTE)&(lpPMIconData->iResource),
            lpPMIconData->dwResSize,
            TRUE,
            lpPMIconData->dwVer
        );
```

Additional reference words: 3.10 3.50
KBCategory: kbui
KBSubcategory: UsrDde

Win32 Software Development Kit Buglist

PSS ID Number: Q95804

The information in this article applies to:

- FastTips for the Microsoft Win32 Software Development Kit (SDK) versions 3.5, 3.51, and 4.0

INSTRUCTIONS

Microsoft FastTips is available 24 hours a day, 7 days a week, from a touch-tone telephone. To order items from this catalog, first select the items you want to receive, noting the five- to six-digit number of the Item ID shown below for each item, and then:

- Dial the toll-free FastTips number (800) 936-4300.
- When prompted, select the Win32 Software Development Kit.
- Press one (1) on your phone keypad to select Express Order Service.
- When prompted, select the delivery method, fax.
- When prompted, enter your three-digit area code and seven-digit fax number on your phone keypad.
- When prompted, enter the number of the Item ID and press #, for up to five items.

When finished, simply hang up. If you have problems receiving a fax, please call (206) 635-3105.

ARTICLE LISTING

ITEM ID	ARTICLE TITLE	PAGES
Q 121907	BUG: Win32 SDK Ver. 3.5 Bug List for Win32 SDK and Win32 API	4
Q 122048	BUG: Win32 Ver 3.5 SDK Bug List at Release - Subsystems & WOW	2
Q 122679	BUG: Win32 SDK Version 3.5 Bug List - OLE	2
Q 122681	BUG: Win32 SDK Version 3.5 Bug List - WinDbg Debugger	4
Q 125872	BUG: GetKerningPairs Sometimes Fails on Win32s Version 1.2	1
Q 126865	BUG: Bad Characters in 32-bit App on Win32s on Russian Windows	1
Q 128701	BUG: CreateDC Does Not Thunk DEVMODE Structure Correctly	1
Q 129861	BUG: Pressing SHIFT+ESC Doesn't Generate WM_CHAR on Windows 95	1
Q 130138	BUG: Win32s 1.25a Bug List	2
Q 130611	BUG: Using WM_SETREDRAW w/ TreeView Control Gives Odd Results	1
Q 130691	BUG: ESC/ENTER Keys Don't Work When Editing Labels in TreeView	1
Q 130699	BUG: SNMP Service Produces Bad "Error on getproc(InitEx) 127"	2
Q 130717	BUG: Console Applications Do Not Receive Signals on Windows 95	1
Q 130860	BUG: FindFirstFile() Does Not Handle Wildcard (?) Correctly	1
Q 131416	BUG: WNetGetUniversalName Fails Under Windows 95	2

End of listing.

Additional reference words: 3.50 4.00 95
KBCategory: kbref kbtlc
KBSubcategory: GenSDK

Win32 Subsystem Object Cleanup

PSS ID Number: Q89290

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

SUMMARY

=======

The Win32 subsystem guarantees that all Win32 objects owned by a process will be freed when an application terminates. To accomplish this, the Win32 subsystem keeps track of who owns these objects; it also keeps a reference count. Reference counts are used when the object is owned by more then one process. For example, a memory mapped file can be used to provide interprocess communication, where more than one process would own that object. The subsystem must make sure that the reference count is zero before the object can be freed.

Freeing of Win32 objects can occur at different times. In general, it occurs at process termination, but for some objects, it occurs at thread termination.

NOTE: When running Win32-based applications with Windows 3.1 using the Win32s environment, it is the responsibility of the Win32-based application to ensure that all allocated GDI objects are deleted before the program terminates. This is different from the behavior of the application with Windows NT. With Windows NT, the GDI subsystem cleans up all orhphaned GDI objects. Because there is no GDI subsystem with Windows 3.1, this behavior is not supported.

MORE INFORMATION

================

At process or thread termination, the Win32 subsystem searches its lists to find objects owned by this process or thread. Those that are owned by the terminating process or thread and whose reference counts will be set to zero when the process or thread is fully terminated will be freed.

The freeing of objects is slightly different for Win32-based applications running under Win32s on Windows 3.1. The 16-bit objects (GDI objects, windows, global memory, etc.) follow the same clean-up rules as Windows-based applications do under Window 3.1. The 32-bit objects, such as memory allocated via VirtualAlloc(), shared memory via mapped file I/O, 32-bit modules, thunks allocated on the fly (for hook procedures, wndprocs etc.) are all handled by Win32s and freed at process termination.

The following is a list of Win32 objects. Note that it may not be complete.

BASE: console, event, file (including file mapping), mutex, semaphore, thread, process, pipe (including named pipes)

GDI: device context (DC), bitmap, pen, brush, font, region, palette

USER: window, cursor, icon, menu, accelerator table, desktop,
DDE communication objects, DDE conversation objects, dialog

Additional reference words: 3.10
KBCategory: kbprg
KBSubcategory: SubSys

Win32s 1.30a Bug List at the Time of Its Release

PSS ID Number: Q138234

The information in this article applies to:

- Microsoft Win32s version 1.30a

SUMMARY

=======

This article lists the known bugs in Win32s version 1.30a at the time of its release.

MORE INFORMATION

================

- Incorrect context at EXIT_PROCESS_DEBUG_EVENT.

- DEBUG_EVENT.RIP_INFO not supported.

- Progman gets restored when debugger app exits.

- Problem with journaling hooks on Win32s when using MSTest.

- OLE: need try/except at 32-bit entrypoints.

- EXCEPTION_RECORD: The # of params and Info array always zero.

- Frame: Access Violation in PackFindData@8 (FindFirstFile looking for arrow.pcx).

- Using StartDoc, the return value is correct but the printer does not output the document.

- CVW32s: int 3 disables receipt of bp for DbgBreakpoint.

- CVW32s: Machine deadlock w/debugger, div0.exe & FatalAppExit.

- CVW32s: Ubrowse.exe runs on NT, but not Win32s example.

- cvw/win32s: restart of mfc app blows.

- cvw/w32s: exception; procterm; l; g; async stop doesn't stop.

- EM_GETWORDBREAKPROC return code incorrect. If an application sends the EM_GETWORDBREAKPROC message to an edit control, it will receive the address of a 32->16 callback thunk with no code on the other end of the stub.

- Dialog procs should return BOOL not DWORD.

- The DDE message thunks force the fAckReq bit to 1 if WM_DDE_DATA is sent with it set to 0. This forces the client to post the

WM_DDE_ACK message so that the thunk layer can know when to free up data allocated for thunking the WM_DDE_DATA message. The server isn't expecting the ACK in this case. The thunk layer will recognize the forced ACK, however the code to consume it has not been implemented yet. Problems so far unseen but still possible.

- Int 3 cannot be trapped via SEH on Win32s.

- Win32's CreateFile() opens files in cooked mode as opposed to raw mode.

- Some time functions, such as ctime(), return values that are off by nine hours. This happens when the TZ environment variable is not defined. When TZ is defined and set to 0, these time functions return the correct value. The time offset is determined by the TZ environment variable, and when it is not defined a default of -9 is used.

- MSTest script Mantapp.mst detects stack overflow when test runs out of memory the second time.

- _lopen - Wrong error code when file does not exist. _lopen failes with error code 87 instead of error code 2 (ERROR_FILE_NOT_FOUND). Win32s is unable to determine from the return value provided by Windows if the call failed due to the file not being there.

- Cannot do ReadProcessMemory() on memory that has hardware bp set on it.

- MFC Speakn sample won't run on Win32s.

- UT Diagram in Programmers Reference is incorrect.

- CRT: getdcwd/getcwd does not work on Win32s.

- Exception in DLL termination routine causes the system to quit.

- PlayMetaFileRecord/EnumMetaFile contains incorrect lpHTable.

- Windbg assertion on HOST machine.

- The .pif files are not supported.

- PlaySound() with SND_RESOURCE.

- File mapping rounded to a whole number of pages, a multiple of 256 instead of 4096. The workaround is to use SetFilePointer() and SetEndOfFile() after closing the mapping handles, but before closing the file.

- chdrive() and SetCurrentDirectory() fail on PCNFS drive.

- Microsoft Excel bapco MSTest script does not run to completion.

- Dynamic dialog boxes under Win32s. An ANSI Win32 application loads a dialog box resource directly from its executable via LoadResource(). Traverses the dialog box template and renders its controls into another dialog box. The application then displays the dialog box which

re-positions some of its controls during WM_INITDIALOG. In Windows NT, the controls rearrange as intended whereas in Win32s, they appear misaligned by 50 pixels or so.

- GetExitCodeProcess() does not return exit codes for 16-bit Windows-based applications.

- Limitation in how NCB memory needs to be allocated on Win32s. The Win31 netbios VxD expects the memory to be allocated from the Win31 memory space to lock that specific memory page. If the memory is allocated from somewhere else such as from the Win32s sparse area, it will fail. In this specific case, allocate a buffer using GlobalAlloc(), memcpy in it the data retrieved, and call Netbios.

- biSizeImage field of BITMAPINFOHEADER is zero. GetDIBits() is not placing the correct value for this field into the buffer.

- Only the first CBT hook gets messages.

- Registry functions return incorrect result code. Registry functions such as RegCreateKey() return incorrect results upon failure. The Windows 3.1 return code is returned, not the Windows NT return code. The functions that return correct results are: RegQueryValueA() and RegQueryValueExA()

- General protection (GP) fault in WinExec() 16-bit application context.

- Bad handles from CreateProcess16() if the child doesn't yield.

- GlobalReAlloc(x,y,GMEM_MOVEABLE) returns wrong handle type.

- GMEM_SHARE memory must be freed explicitly.

- RestartThread() writes to debuggee stack.

- The memory for CallBack frames is allocated once. The allocation size is 64K. In most cases this is too large, and in some extreme cases this 64K may be exhausted.

- CreateDirectory() handles errors differently between Windows NT and Win32s. Under Windows NT, all failures are handled correctly. For example, directory already exists, no space left on device, and so on. Under Win32s, all errors are handled identically (error #5, "Access is denied").

- SetCurrentDirctory(): different error codes on Windows NT and Win32s. SetCurrentDirectory() works differently under Windows NT 3.5 and Win32s. For example, if there is no floppy disk in drive B, and SetCurrentDirectory() is called with "b:\", under Windows NT error #21 is returned, "The device is not ready," while under Win32s, error #161, "The specified path is invalid" is returned.

- signal(SIGINT, SIGABRT, SIGTERM): need instance data.

- FindText() leaks memory and may cause a general protection (GP) fault.

- spawnl doesn't pass parameters to an MS-DOS-based application.

- Win32s does not support forwarded exports.

- module management APIs missing ANSI to OEM translation.

- PlaySound() w/ SND_NOWAIT.

- FormatMessage() doesn't set last error.

- Createfile(), when used on a write-protected floppy disk, fails. GetLastError() returns 2 instead of 19.

- Win32app cannot be browsed if the same application is running.

- Setup doesn't create backup files for some of the OLE files.

- winhlp32:Compare macro does not display windows side by side.

- MEASUREITEMSTRUCT and DRAWITEMSTRUCT are missing fields in menus.

- HeapValidate() fails on a new heap. HeapValidate() is not supported in the retail version of Win32s. GetLastError() returns ERROR_CALL_NOT_IMPLEMENTED. HeapValidate() is supported only in the debug version of Win32s and in both the retail and debug versions of Windows NT.

  Other new heap APIs that are not supported in Win32s are:

  HeapCompact()
  HeapCreateTags()
  HeapExtend()
  HeapLock()
  HeapQueryTag()
  HeapSummary()
  HeapUnlock()
  HeapUsage()
  HeapWalk()

  All these APIs, including HeapValidate, are documented as not supported on Win32s and Windows 95.

- winhlp32:delete of annotation in popup causes unh exception.

- winhlp32:PopupContext macro produces incorrect error message.

- winhlp32: PC macro with non-help file.

- winhlp32:PopupID with non-existing context string.

- A file opened as GENERIC_READ only can still be written to.

- winhlp32 doesn't play .avi files.

- winhlp32:new highlight feature is not consistent.

- winhlp32:System error on write protected medium.

- winhlp32:bitmap in Shed.hlp does not print correctly.

- winhlp32:winword produces error when attempting to print from a
  secondary window.

- winhlp32:Clicks go through File Manager.

- MPLAY32.EXE of Windows NT 3.1 does not run on Win32s.

- winhlp32:Window title does not change back.

- winhlp32:kicked out of help after opening non-helpfile.

- SearchPath() doesn't find file if Path is empty. NULL for lpszPath
  is fine, but an lpszPath = "" will not work the same as on Windows NT.

- Wrong exception reported for overflow exception.

- VerLanguageNameA() not exported by version.dll.

Additional reference words: 1.30a
KBCategory: kbprg kbbuglist
KBSubcategory: W32s

Win32s 1.30c Bug List at the Time of Its Release

PSS ID Number: Q148862

The information in this article applies to:

- Microsoft Win32s version 1.3c

SUMMARY

=======

This article lists all the known bugs in Win32s version 1.30c at the time of its release.

MORE INFORMATION

================

- Incorrect context at EXIT_PROCESS_DEBUG_EVENT.
- DEBUG_EVENT.RIP_INFO is not supported.
- Progman is restored when the debugger application exits.
- Problem with journaling hooks on Win32s when using MSTest.
- OLE: need try/except at 32-bit entrypoints.
- EXCEPTION_RECORD: The number of params and Info array is always zero.
- Frame: Access Violation in PackFindData@8 (FindFirstFile looking for Arrow.pcx).
- Using StartDoc, the return value is correct but the printer does not print the document.
- CVW32s: int 3 disables the receipt of bp for DbgBreakpoint.
- CVW32s: Machine deadlock with the debugger, div0.exe, and FatalAppExit.
- CVW32s: Ubrowse.exe runs on Windows NT but not Win32s example.
- cvw/win32s: restart of MFC application fails.
- cvw/w32s: exception; procterm; l; g; async stop doesn't stop.
- EM_GETWORDBREAKPROC return code is incorrect. If an application sends the EM_GETWORDBREAKPROC message to an edit control, it will receive the address of a 32->16 callback thunk with no code on the other end of the stub.
- Dialog procs should return BOOL not DWORD.
- The DDE message thunks force the fAckReq bit to 1 if WM_DDE_DATA is sent with it set to 0. This forces the client to post the WM_DDE_ACK message

so that the thunk layer can know when to free up data allocated for thunking the WM_DDE_DATA message. The server isn't expecting the ACK in this case. The thunk layer will recognize the forced ACK, however the code to consume it has not been implemented yet.

- Int 3 cannot be trapped by way of SEH on Win32s.

- Win32's CreateFile() opens files in cooked mode as opposed to raw mode.

- Some time functions, such as ctime(), return values that are off by nine hours. This happens when the TZ environment variable is not defined. When TZ is defined and set to 0, these time functions return the correct value. The time offset is determined by the TZ environment variable, and when it is not defined, a default of -9 is used.

- Microsoft Test script Mantapp.mst detects stack overflow when the test runs out of memory the second time.

- _lopen gives the wrong error code when the file does not exist. _lopen fails with error code 87 instead of error code 2 (ERROR_FILE_NOT_FOUND). Win32s is unable to determine from the return value provided by Windows if the call failed due to the file not being there.

- Cannot run ReadProcessMemory() on memory that has hardware bp set on it.

- UT Diagram in the Programmers Reference is incorrect.

- CRT: getdcwd/getcwd does not work on Win32s.

- An exception in the DLL termination routine causes the system to quit.

- PlayMetaFileRecord/EnumMetaFile contains incorrect lpHTable.

- A Windbg assertion occurs on the HOST machine.

- PlaySound() with SND_RESOURCE doesn't work correctly.

- File mapping rounded to a whole number of pages (a multiple of 256 instead of 4096). The workaround is to use SetFilePointer() and SetEndOfFile() after closing the mapping handles, but before closing the file.

- chdrive() and SetCurrentDirectory() fail on the PCNFS drive.

- Microsoft Excel bapco Microsoft Test script does not run to completion.

- Dynamic dialog boxes under Win32s: An ANSI Win32 application loads a dialog box resource directly from its executable through LoadResource(), traverses the dialog box template, and renders its controls into another dialog box. The application then displays the dialog box, which repositions some of its controls during WM_INITDIALOG. In Windows NT, the controls rearrange as intended, but in Win32s, they appear misaligned by 50 pixels or so.

- GetExitCodeProcess() does not return exit codes for 16-bit Windows-based

applications.

- The CRT function system() does not work the way it did in Windows NT.

- There is a limitation in how NCB memory needs to be allocated on Win32s. The Win31 netbios VxD expects the memory to be allocated from the Win31 memory space to lock that specific memory page. If the memory is allocated from somewhere else such as from the Win32s sparse area, the memory allocation will fail. In this specific case, allocate a buffer using GlobalAlloc(), memcpy in it the data retrieved, and call Netbios.

- biSizeImage field of BITMAPINFOHEADER is zero. GetDIBits() is not placing the correct value for this field into the buffer.

- Only the first CBT hook gets messages.

- Registry functions return incorrect result code. Registry functions such as RegCreateKey() return incorrect results upon failure. The Windows 3.1 return code is returned, not the Windows NT return code. The functions that return correct results are: RegQueryValueA() and RegQueryValueExA()

- General protection (GP) fault in WinExec() 16-bit application context.

- Bad handles from CreateProcess16() if the child doesn't yield.

- GlobalReAlloc(x,y,GMEM_MOVEABLE) returns the wrong handle type.

- GMEM_SHARE memory must be freed explicitly.

- RestartThread() writes to the debuggee stack.

- The memory for CallBack frames is allocated once. The allocation size is 64K. In most cases, this is too large, and in some extreme cases this 64K may be exhausted.

- GetShortPathName() does not check the correctness of the path.

- CreateDirectory() handles errors differently in Windows NT from the way it handles errors in Win32s. Under Windows NT, all failures (directory already exists, no space left on device, and so on) are handled correctly. Under Win32s, all errors are handled identically (error #5, "Access is denied").

- SetCurrentDirctory() gives different error codes on Windows NT from what it gives on Win32s. SetCurrentDirectory() works differently under Win32s. For example, if there is no floppy disk in drive B, and SetCurrentDirectory() is called with "b:\" under Windows NT, error #21 is returned ("The device is not ready") while under Win32s, error #161 is returned ("The specified path is invalid").

- signal(SIGINT, SIGABRT, SIGTERM) needs instance data.

- FindText() leaks memory and may cause a general protection (GP) fault.

- spawnl doesn't pass parameters to an MS-DOS-based application.

- Win32s does not support forwarded exports.

- module management APIs missing ANSI to OEM translation.

- GetIconInfo()'s piconinfo->fIcon member is always set to 1.

- PlaySound() with SND_NOWAIT doesn't work correctly.

- FormatMessage() doesn't set the last error.

- Createfile(), when used on a write-protected floppy disk, fails. GetLastError() returns 2 instead of 19.

- Win32app cannot be browsed if the same application is running.

- Setup doesn't create backup files for some of the OLE files.

- winhlp32:Compare macro does not display windows side by side.

- MEASUREITEMSTRUCT and DRAWITEMSTRUCT are missing fields in menus.

- HeapValidate() fails on a new heap. HeapValidate() is not supported in the retail version of Win32s. GetLastError() returns ERROR_CALL_NOT_IMPLEMENTED. HeapValidate() is supported only in the debug version of Win32s and in both the retail and debug versions of Windows NT.

  Other new heap APIs that are not supported in Win32s are:

  HeapCompact()
  HeapCreateTags()
  HeapExtend()
  HeapLock()
  HeapQueryTag()
  HeapSummary()
  HeapUnlock()
  HeapUsage()
  HeapWalk()

  All these APIs, including HeapValidate, are documented as not supported on Win32s and Windows 95.

- winhlp32:delete of annotation in popup causes unh exception.

- GrayString() with a NULL callback address causes a Fatal Exit in the debug version of Windows.

- SetEnvironment does not handle various errors correctly

- winhlp32:PopupContext macro produces incorrect error message.

- winhlp32:PC macro with non-help file doens't work correctly.

- winhlp32:PopupID with non-existing context string doesn't work
  correctly.

- A file opened as GENERIC_READ only can still be written to.

- CreateFile does not update attributes when file is opened for the
  second time.

- PrintDlg Error - Print Dialog succeeds with NULL hPrintTemplate
  and PD_ENABLEPRINT.

- PrintDlg Error - Print Dialog succeeds with NULL hSetupTemplate
  and PD_ENABLESETUPTEMPLATE

- PrintDlg Error - Print Dialog succeeds with NULL hInstance.

- PrintDlg Error - If the From value is greater than the To value in the
  Print dialog, it still returns SUCCESS instead of an error value.

- PrintDlg Error - If the From or To entry is empty in the Print dialog,
  it still returns SUCCESS instead of an error value.

- winhlp32 doesn't play .avi files.

- winhlp32:new highlight feature is not consistent.

- winhlp32:System error on write protected medium.

- winhlp32:bitmap in Shed.hlp does not print correctly.

- winhlp32:winword produces error when attempting to print from a
  secondary window.

- winhlp32:Clicks go through File Manager.

- Mplay32.exe of Windows NT 3.1 does not run on Win32s.

- winhlp32:Window title does not change back.

- winhlp32:kicked out of help after opening non-helpfile.

- SearchPath() doesn't find file if Path is empty. NULL for lpszPath
  is fine, but an lpszPath = "" will not work the same as on Windows NT.

- VerLanguageNameA() is not exported by Version.dll.

- Low level wave in win32s (WAVEHDR.dwBytesRecorded) is set to zero.

- MoveFile fails on Novell client 4.0 when the source is a local file
  and the target is a remote file.

- SetWindowLong (GWL_USERDATA) causes a warning from the debug version of
  Windows when the window is destroyed

- Setting a word break function on an edit control causes a general

protection (GP) fault.

- Tooltips does not appear in Edit controls

- WinHelp doesn't handle the HELP_TCARD parameter properly.

- WM_CHOOSEFONT_GETLOGFONT does not work correctly.
* #711

Additional reference words: 1.30c bugs buglist
KBCategory: kbprg kbbuglist
KBSubcategory: W32s

Win32s and LAN Manager APIs

PSS ID Number: Q109204

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, 1.15, 1.2, 1.25, and 1.25a

---

NOTE: Win32s provides support for NetBIOS and Windows Sockets.

There is a NETAPI32.DLL shipping with Win32s. However, this doesn't have support for Microsoft LAN Manager application programming interfaces (APIs). This is not a bug because LAN Manager APIs are not supported on Win32s. However, if NETAPI32.DLL is not in the system and an application uses LAN Manager APIs, the application will not load because the loader cannot resolve the entry-points from the nonexistent dynamic-link library (DLL).

One of the solutions is to use Universal Thunks. Using Universal Thunks, calls can be made to 16-bit DLLs. Therefore, the 16-bit NETAPI.DLL that ships with LAN Manager can be accessed by this mechanism.

Sample programs using Universal Thunks are provided in the Win32 SDK and the Visual C++ SDK. The sample on the Win32 SDK is in the MSTOOLS\WIN32S\UT\SAMPLES directory. On the Visual C++ 32-bit Edition CD-ROM, the sample is located in the MSVC32S\WIN32S\UT\SAMPLES directory.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Win32s and Windows NT Timer Differences

PSS ID Number: Q105758

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

Under Windows NT, timers are system objects; as such, they are not owned by an application. SetTimer() can be called from within one application with a handle to a window that was created by a different application. This application would process the WM_TIMER messages in the window procedure. The timer event will continue to occur even after the application that created the timer has terminated. Note that it is fairly uncommon for a Win32-based application to create a timer for another application, but this method does work.

Because Win32s runs on top of Windows 3.1 and shares many of its characteristics, timers are owned by the application that calls SetTimer(). The timer event terminates when the application that owns the timer terminates.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Win32s Device-Independent Bitmap Limit

PSS ID Number: Q115084

The information in this article applies to:

- Microsoft Win32s, versions 1.1, 1.15, 1.2, and 1.25a

Under versions of Win32s prior to 1.25, a device-independent bitmap (DIB) is limited to a size of 2.3 MB. This size was chosen to accomodate a bitmap of 1024 by 768 pixels at 24 bits per pixel.

In Win32s 1.25a, this limit was increased to 1280 by 1024 pixels at 24 bits per pixel.

This limit can cause a variety of problems to occur, such as painting problems with SetDIBitstoDevice() if a larger bitmap is used.

Additional reference words: 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Win32s Is No Longer Limited to 256 Selectors

PSS ID Number: Q147428

The information in this article applies to:

- Microsoft Win32s version 1.2 and later

Prior to version 1.2, Win32s had a limit of 256 selectors that it maintained for selector synthesis and once a selector had been allocated, it could not be reused after freeing it.

When thunking pointers down to the 16-bit side, Win32s allocates selectors on the fly for mapping the 0:32 linear address to the 16:16 address that the 16-bit side could use. These synthesized selectors were saved in an internal table and reused later whenever needed. It was this table that was limited only to 256 selectors in Win32s prior to version 1.20. Since Win32s could reuse these synthesized selectors, they can not be freed either.

However, this problem was eliminated in Win32s version 1.2 and the solution was further enhanced in Win32s version 1.25. Now, the current version of Win32s (version 1.30c) is no longer limited to 256 selectors.

However, it is still possible that a Win32s application could consume all available selectors in the system and potentially bring Windows 3.x to a halt, but such a scenario rarely occurs and is probably specific to a given application.

Additional reference words: 1.20 256 selector synthesis fix kbinf
KBCategory: kbprg
KBSubcategory: w32s

Win32s Message Queue Checking

PSS ID Number: Q97918

The information in this article applies to:

- Microsoft Win32s versions 1.1 and 1.2

Win32-based applications that are designed to run with Win32s need to check the message queue through a GetMessage() or PeekMessage() call to avoid locking up the system. With Windows NT this is not a problem, because the input model is desynchronized. That is, each thread has its own input event queue rather than having one queue for the entire system. In a synchronous input model, one application can block all of the others by allowing the single system queue to fill up with its messages. With Windows NT, an application that lets its input queue fill up will not affect other applications.

For more information, please see chapter 3 of the "Win32s Programmer's Reference".

Additional reference words: 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Win32s NetBIOS Programming Considerations

PSS ID Number: Q104314

The information in this article applies to:

- Microsoft Win32s version 1.0, 1.1, and 1.2

This article addresses common questions about NetBIOS programming under Win32s.

Under Windows 3.1, you normally do not issue a RESET command due to the common name table. When running your Win32-based application under Windows version 3.1, you may issue a NetBIOS RESET command. Win32s keeps an internal list of all the names added in the Windows 3.1 NetBIOS name table by Win32 processes. A RESET on Win32s mimics the RESET on Windows NT by clearing the names added by that process from the system-wide name table. As a result, issuing a RESET command as the first NetBIOS command (as required by Windows NT) on Win32s does not clear out all of the names.

The Windows 3.1 NetBIOS VxD expects memory allocated for the NetBIOS Control Block (NCB) and the data buffer (NCB.ncb_buffer) to be allocated with GlobalAlloc(). The VxD will lock the specified memory page. If a Win32-based application running under Win32s passes the Netbios() command virtual memory, Netbios() will return error 0x22, indicating that there are too many commands outstanding. On Win32s, each piece of memory that might pass through Netbios() must be allocated with GlobalAlloc().

The Win32s NetBIOS thunk layer translates the ncb_buffer pointer in the NCB itself. The translation back from a 16-bit to original 32-bit pointer is done for asynchronous commands in the Win32s private post routine. A problem might occur when an application checks the NCB and finds that the netbios() command is completed (ncb_retcode == NRC_GOODRET), but the Win32s post routine was not called yet. The ncb_buffer has not been translated back. The best way to avoid problems is to define and use a post routine. At that point you are sure that the netbios() command is completed and the NCB is correct. If you don't want to use a post routine, you should make sure that the command was completed by checking the the ncb_retcode field and verifying that the ncb_buffer pointer is a 32-bit pointer.

There is also no need to page lock the NetBIOS post routine code under Win32s. NetBIOS post routine on Win32s is not called at interrupt time. The post routine is called in the context of the process, and therefore there is no need to page lock the post routine code.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Win32s OLE 16/32 Interoperability

PSS ID Number: Q123422

The information in this article applies to:

- Microsoft Win32s version 1.20

The OLE support provided in Win32s version 1.2 provides full 16/32 interoperabililty for local servers (EXE servers). Therefore, you can embed a 32-bit object implemented by a local server in a 16-bit container and vice-versa.

There is no built-in support in Win32s for 16/32 interoperability for in-process servers (DLL servers). However, you can use Universal Thunks to load a 16-bit DLL in the context of a 32-bit process. This allows you to embed a 16-bit object implemented by a DLL server in a 32-bit container. However, it is quite complicated to write this code because:

- Any OLE interface has a hidden "this" pointer which you must handle in your thunking code.
- OLE uses callbacks. If your 32-bit container calls IDataObject:DAdvise, then your 16-bit server may call back into the 32-bit side with the Advise interface. Your thunking code will have to handle this type of conversation.

NOTE: Embedding a 32-bit object implemented by an in-process server in a 16-bit container is supported under Windows NT 3.5. However, this functionality may not work correctly for your in-process server because IDispatch and any custom interfaces do not work.

Additional reference words: 1.20
KBCategory: kbole
KBSubcategory: W32s

Win32s Translated Pointers Guaranteed for 32K

PSS ID Number: Q100833

The information in this article applies to:

- Microsoft Win32s versions 1.0, 1.1, and 1.2

SUMMARY

=======

Translated pointers are guaranteed to be valid only for 32K, rather than 64K, which selectors are usually limited to. This limitation is for performance reasons.

Selectors are tiled every 32K. A 0:32 pointer can be quickly translated into a 16:16 pointer, which will be valid for a minimum of 32K. In other words, the offset portion of the 16:16 pointer is not guaranteed to be 0 (zero) when translated. As a result, even though the translated selectors have a limit of 64K, the offset passed to the 16-bit side may be as large as 32K-1.

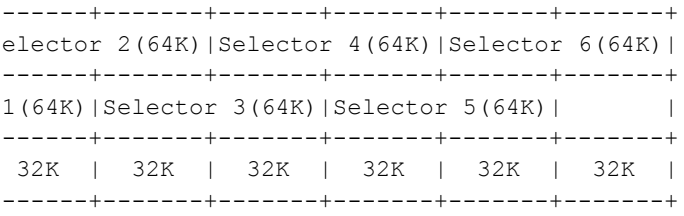
Selectors are created on a 32K alignment so that if you pass several pointers to the same range, the Universal Thunk (UT) uses the same selector. Selectors are freed when application terminates.

The alternative is to create a selector for each and every translation, which is very slow.

MORE INFORMATION

================

For any given address, there are two selectors that point to it, but only one has a limit less than 32K:



Under Win32s, 16-bit and 32-bit applications share the same global data space; therefore, it is possible to share a buffer of up to 64K in size with a far pointer or more than 64K with a huge pointer by doing the following:

1. Do a GlobalAlloc() on the 32-bit side. Be sure to use GMEM_MOVEABLE.
2. Copy the data.
3. Send the handle to the 16-bit side.
4. Get a pointer to the data on the 16-bit side by using GlobalLock().

When you pass a pointer to a block that was allocated via GlobalAlloc()

from the 32-bit side, it costs no selectors. The translated pointer is valid until the memory is freed.

Additional reference words: 1.00 1.10 1.20
KBCategory: kbprg
KBSubcategory: W32s

Win32s Version 1.25a Limitations

PSS ID Number: Q131896

The information in this article applies to:

- Microsoft Win32s, version 1.25a

SUMMARY
=======

The following lists the limitations of Win32s, not including the complete list of unsupported Win32 APIs. For information on which Win32 APIs are and are not supported under Win32s, please see the "Win32 Programmer's Reference" or the WIN32API.CSV file.

MORE INFORMATION
================

- Thread creation is not supported.

- Win32s uses the Windows version 3.1 nonpreemptive scheduling mechanism.

- 32-bit processes should be started from a 16-bit application through WinExec() or Int 21 4B. LoadModule() does not start a 32-bit process.

- Win32-based applications cannot use the EM_SETHANDLE or EM_GETHANDLE messages to access the text in a multiline edit control. These messages allow the sharing of local memory handles between an application and USER.EXE. In Win32s, an application's local heap is 32-bit, so USER.EXE cannot interpret a local handle. To read and write multi-line edit control text, an application should use WM_GETTEXT and WM_SETTEXT.

  Text for multi-line edit controls is stored in the 16-bit local heap in DGROUP, which is allocated by Windows version 3.1 for the WIN32S.EXE stub loaded before each Win32-based application is loaded. This means that the size of edit control text is limited to somewhat less than 64K.

- The CBT hook thunks do not copy back the contents of structures, so any changes are lost.

- Win32-based applications should not call through the return from SetWindowsHook(). SetWindowsHook() returns a pointer to the next hook in the chain. To call this function, the application should call DefHookProc(), and pass a pointer to a variable where this address is stored. Because Win32s simulates the old hook API in terms of the new, SetWindowsHook() actually passes back a hook handle, not a function pointer.

- Resource integer IDs must be 16-bit values.

- PostMessage() and PeekMessage() do not thunk structures. These functions do not allocate space for repacking structures, because there is no way to know when to free up the space.

- Private application messages should not use the high word of wParam. Win32s uses Windows to deliver window messages. For unknown messages, wParam is truncated to make it fit into 16-bits. Therefore, Win32-based applications should not put 32-bit quantities into the wParam of privately-defined messages, which are unknown to the thunk layer.

- After calling FindText(), an application cannot directly access the FINDREPLACE structure. When an application calls the common dialog function FindText(), it passes a FINDREPLACE structure. The FindText() dialog communicates with the owner window through a registered message, in which the lParam points to the structure. The thunks repack this structure in place, so that unless the application is processing the registered message, it should not access the structure.

- Subclassing a window that owns a FindText() common dialog does not work. The FindText() thunk repacks the FINDREPLACE structure in place. This means that 32-bit window procedures cannot subclass 16-bit owners of FindText() dialogs without likely trashing four bytes beyond the end of the structure, because the 32-bit FINDREPLACE is four bytes bigger than the 16-bit version.

  For 32-bit owners of the dialog box, there are also problems. Whenever the structure might be referenced by 16-bit code, it needs to be in the 16-bit format. However, when the FindText() dialog box terminates, the structure needs to be in 32-bit format so that the application can read the final values. The dialog box indicates to its owner that it is about to be destroyed by sending the commdlg_FindReplace message with the FR_DIALOGTERM bit set in the Flags field of the FINDREPLACE structure. After sending this message, the FindText() dialog box does not reference the structure anymore, so the commdlg_FindReplace thunk leaves it in 32-bit format.

  The problem occurs when the owner has been subclassed. Once the message has moved to the 32-bit side, the structure will not be reconverted to 16-bits. Suppose the dialog box owner was subclassed by a 16-bit window procedure, which was in turn subclassed by a 32-bit window procedure. When a message is sent to the owner, it will first go to 32-bits, then to 16-bits, then back to 32-bits to the original window procedure:

  1. 16->32 message sent to 32-bit subclasser (struct repacked to 32-bits)
  2. 32->16 message sent to 16-bit subclasser (struct not repacked)
  3. 16->32 message sent to 32-bit original WndProc (struct not repacked)

  The 16-bit subclasser cannot interpret the message parameters because they are in 32-bit format. Between steps 2 and 3, the structure is not repacked because it's already in 32-bit format.

  Therefore, when a Win32-based application calls FindText(), allocate the following:

  1. A structure that contains the htask of WIN32S.EXE.

  2. The original 32-bit FINDREPLACE pointer.

3. A count of the number of times the structure has been thunked without returning.

4. A 16-bit format copy of FINDREPLACE.

The structure is repacked into the 16-bit version and this copy is passed to Windows. The count is initially 0. When the commdlg_FindReplace message is sent in either direction (16->32 or 32->16), repack the structure on the stack. If it was originally 32-bit, increment the count. When returning in either direction, unpack the structure and, if originally 32-bit, decrement the count.

If the Flags field of the structure has the FR_DIALOGTERM bit set, the structure is originally 32-bit, and the count goes to 0 on a return, then repack the structure back to the original 32-bit space.

- DDE messages are always posted, not sent, except for two: WM_DDE_INITIATE and WM_DDE_ACK (responding to the former). The sent form of WM_DDE_ACK is different from the posted form. When sent, no thunking of lParam is necessary when going between 16- and 32-bit format. However, when posting, the lParam parameter is repacked as two DWORDS into private memory, allocated by COMBO.DLL (or by USER.DLL in Windows NT). The thunk layer makes this distinction through the InSendMessage() function. Therefore, a Win32-based application should not do anything in the processing of a WM_DDE_INITIATE message that would cause the return code from InSendMessage() to be FALSE.

- The following DDE messages are handled differently by applications under Windows and Win32:

  WM_DDE_ACK
  WM_DDE_ADVISE
  WM_DDE_DATA
  WM_DDE_EXECUTE
  WM_DDE_POKE

  Because of the widening of handles, there is not enough space in the Win32 message parameters for all the Windows data. Win32-based applications are required to call helper functions that package the data into memory (DDEPACK structure) referenced by a special handle. Win32s implements the helper functions and allocates the DDEPACK structure on behalf of 16-bit code when necessary.

  As with other memory handles shared through DDE, there are rules concerning who frees the memory allocated by these helper routines.

```
lParam = PackDDElParam(...)
if (PostMessage(...,lParam))
   receiving window has obligation of freeing memory
   lParam now invalid for this process
else
   FreeDDElParam(...,lParam)
```

  A Win32s hook procedure that has called CallNextHookProc() should not use the lParam handle as the later's return code indicates that the

message has been processed. The hooks that could possibly process a DDE message are:

```
WH_DEBUG
WH_HARDWARE
WH_KEYBOARD
WH_MOUSE
WH_MSGFILTER
WH_SYSMSGFILTER
```

In each case, if the hook proc (and therefore CallNextHookEx()) returns a non-zero value, the message was either discarded or processed. Windows-based procedures should not use the lParam of a DDE message after handing it off to any other message API because they cannot know if the message has been processed, so they must assume that it has been.

- WDEB386 does not support FreeSegment() for 32-bit segments. Win32s Kernel Debugger support depends on it.

- The maximum size of shared memory is limited by Windows memory.

```
CreateFileMapping(hFile,
                  lpsa,
                  fdwProtect,
                  dwMaximumSizeHigh,
                  dwMaximumSizeHigh,
                  lpszMapName)

lpsa              - Ignored.
dwMaximumSizeHigh - Must be zero.


MapViewOfFile(hMapObject,
              fdwAccess,
              dwOffsetHigh,
              dwOffsetLow,
              cbMap)

dwOffsetHigh - Must be zero.
```

- SetClipbrdData() must be used only with a global handle. Otherwise, the data can't be accessed by other applications.

- Win32s supports printing exactly as Windows version 3.1 does. Win32s does not add beziers, paths, or transforms to GDI; to allow applications to use these features on PostScript printers, you must call the Escape function with the appropriate escape codes.

  Applications link to Windows version 3.1 printer drivers through LoadLibrary() and GetProcAddress(), which have special support for this purpose. There is no general mechanism allowing a Win32-based application to link to a 16-bit DLL.

- Win32s does not support these escape codes:

```
BANDINFO                ;24
GETSETPAPERBINS         ;29
ENUMPAPERMETRICS        ;34
EXTTEXTOUT              ;512
SETALLJUSTVALUES        ;771
```

- Integer atoms must be in the range 0-0x3FFF. This restriction is necessary for the current implementation of the window properties API thunks: SetProp(), GetProp(), RemoveProp(), EnumProps(), and EnumPropsEx(). Integer atoms above 0x4000 are rejected by the thunk layer.

- Arrays must fit in 64K after converting to 16-bit format. An array passed to a function such as SetCharABCWidths() or Polyline() must fit within 64K after its elements have been converted to their 16-bit form. Its 32-bit form may be bigger than 64K.

- All Windows version 3.1 APIs that return void, return 1 to the Win32-based application.

  You can simulate a boolean return value by validating the API parameters and returning FALSE if one is bad. However, you can't always do complete validation, and if the API is called, you must assume it succeeded unless there is a method to verify whether or not it succeeded.

- Win32 child window IDs must be sign-extended 16-bit values. This excludes the use of 32-bit pointer values as child IDs.

- When calling PeekMessage(), a Win32-based application should not filter any messages for Windows internal window classes (button, edit, scrollbar, and so on). The messages for these controls are mapped to different values in Win32, and checking for the necessity of mapping is time-consuming.

- The dwThreadId parameter in SetWindowsHookEx() is ignored. The dwThreadId is translated to hTask in Windows 3.1. There's a bug in Windows version 3.1 where if hTask!=NULL, the call may fail.

- CreateWindowEx() has a DWORD dwExStyle parameter. In Windows 3.1 the hiword of dwExStyle is cleared as a protection against garbage in the hiword before it's used. Win32s passes CreateWindowEx() to the 16 bit CreateWindowEx() and the hi 16 bits in the dwExStyle are lost.

- Floating point (FP) emulation by exception cannot be performed in 16-bit applications. When tasks are switched between applications, the CR0-EM bit state is not preserved in order to support 32-bit application FP emulation by exception without breaking the existing 16-bit applications that use FP instructions. The CR0-EM bit is assumed to be cleared during execution of 16-bit application FP instructions. Upon executing a 16-bit application FP instruction, the bit is cleared and reset when switching back to a 32-bit application. The CR0-EM bit management is done in the Win32s VxD, thus disabling the possibility of getting an int 7 exception just by setting the CR0-EM bit in a 16-bit application.

- EndDialog() nResult parameter is sign-extended. Applications specify the

return value for the DialogBox() function by way of the nResult parameter to the EndDialog() API. This parameter is of type int, which is 32-bit in Win32s. However, this value is thunked through to the Windows version 3.1 EndDialog() API, which truncates it to a 16-bit value. Win32s sign-extends the return code from DialogBox().

- GetClipBox() returns SIMPLEREGION(2) and COMPLEXREGION(3). Because Windows NT is a preemptive multitasking system, GetClipBox() on Windows NT never returns SIMPLEREGION(2). The reason for this is that between the time the API was called and the time the application gets the result, the region may change. Win32s can return both SIMPLEREGION(2) and COMPLEXREGION(3).

- PeekMessage() filtering for posted messages (hWnd==-1) is not supported. The hWnd is replaced with NULL.

- Message queue length is limited to Windows default: 8 or whatever length was set by DefaultQueueSize=n in the WIN.INI file. This limit may be increased in the future to a larger size, but there will always be a limit.

- GetFileTime() and SetFileTime() process only the lpLastWriteTime parameter and return an error if this parameter is not supplied. In the DEBUG version, supplying the other parameters causes a warning message to be displayed.

- The precision of the time of a file is two seconds (MS-DOS limitation).

- CreateProcess has the following limitations:

  - fdwCreate - Only DEBUG_PROCESS and DEBUG_ONLY_THIS_PROCESS are supported.

  - Process priority is always NORMAL.

  - lpsaProcess, lpsaThread - Security information ignored.

- Always use device-independent bitmaps for color bitmaps. Win32s supports the four Win32 device-dependent bitmap APIs. These are device-dependent in the sense that the bitmap bits are supplied without a color table to explain their meaning.

  CreateBitmap
  CreateBitmapIndirect
  GetBitmapBits
  SetBitmapBits

  These are well defined and fully supported for monochrome bitmaps. For color bitmaps, these APIs are not well defined and Win32s relies on the Windows display driver for their support. This means that an application cannot know the format of the bits returned by GetBitmapBits() and should not attempt to directly manipulate them. The values returned by GetDeviceCaps() for PLANES and BITSPIXEL and the values returned by GetObject() for a bitmap do not necessarily indicate the format of the bits returned by GetBitmapBits(). It is possible for the GDI DIB APIs to

be unsupported on some displays. However, it is now rare for display drivers to not support DIBs. The one case where you may encounter a lack of DIB support is with printer drivers, which may not support the GetDIBits() API, though most do support the SetDIBits() API.

Win32s does not transform the bits in any way when passing them on to a Windows version 3.1 API. When running an application that creates a device-dependent bitmap via CreateBitmap() or CreateBitmapIndirect(), be aware that the bits it is passing in may not be in the right format for the device. Windows NT takes care of this by treating the bits as a DIB whose format is consistent with the PLANES and BITSPIXEL values; but Win32s simply passes them through.

- GetPrivateProfileString() and GetProfileString() return an error when the lpszSection parameter is NULL. Under Windows NT, they give all the sections in the .INI file.

- String resources are limited to a length of 255, as they were in Windows version 3.1.

- TLS locations are the same in all processes for a specific DLL. This is because Win32s does not support per-instance data for DLLs. The TLS locations are unique per DLL. Each DLL should call TlsAlloc() only once if it is runing on Win32s. The index returned will be valid for all Win32 processes.

- GlobalCompact() is thunked through to Windows version 3.1 GlobalCompact(). This API has no effect on memory allocated through VirtualAlloc(), which does not come from the Windows global heap.

- GetVolumeInformation() does not support the Volume ID.

- GetFileInformationByHandle() create time and access time are always 0 (MS-DOS limitations). The volume id, file index low/high are also 0 (Win32s limitations). This affects the CRT fstat() as well.

- CreatePolyPolygonRgn() requires a closed polygon, as it does under Windows version 3.1. If the polygons are not closed, the Windows NT call closes the polygons for you. In Windows version 3.1 or Win32s, if the polygons are not closed, the call does not create the region correctly, or it returns an error for an invalid parameter.

- Win32s does not support the DIB_PAL_INDICES option for SetDIBits(). It will be supported in a future release. DIB_PAL_PHYSINDICES and DIB_PAL_LOGINDICES are not supported either.

- The WH_FOREGROUNDIDLE hook type is not supported. Windows version 3.1 does not provide the necessary support.

- The brush styles BS_DIBPATTERNPT and BS_PATTERN8X8 are not supported and cause an error to be returned.

- The hFile in DLL and PROCESS DEBUG events is not supported in Win32s because there is no support for duplicating file handles between processes (basically an MS-DOS limitation).

There are two way to access the image bytes: ReadProcessMemory() or open the file in compatibility mode using the name provided in lplpImageName.

- Under Windows NT, NetBIOS keeps a different name table for each process. On Win32s, there is only one NetBIOS name table for the system. Each name (group or unique) added by a process is kept in a doubly-linked list. Through NCBRESET or by destroying the process, you delete all the names that were added by the process. Win32s does not implement the full NetBIOS 3.0 specification, which has a separate name table per process.

- When calling 32-bit code from 16-bit code with UT (for example, from an interrupt routine), the stack must be at least 10K. The interrupt routine must assure that the stack will be big enough.

- GetThreadContext() and SetThreadContext() can be called only from within an exception handler or an exception debug event. At all other times, these functions return FALSE and the error code is set to ERROR_CAN_NOT_COMPLETE.

- CreateProcess() PROCESS_INFORMATION is not supported for 16-bit applications. A valid structure must be passed to CreateProcess(), but the function fills all the fields with NULLs and zeroes.

- Win32s does not support the Windows NT event mechanism, therefore the ncb_event field in NCB structure is not supported.

- CreateFileMapping() does not support SEC_NOCACHE or SEC_NOCOMMIT. The call fails with ERROR_INVALID_PARAMETER.

- WaitForDebugEvent() does not fully support the dwTimeout parameter. If the parameter is zero, WaitForDebugEvent() behaves the same as under Windows NT. Otherwise, the parameter is treated as if it were INFINITE. However, the function returns if any messages arrive. If a message arrives, the return value is FALSE. The calling process should call SetLastError(0) before calling WaitForDebugEvent() and examine GetLastError() if WaitForDebugEvent() returns FALSE. If the error is zero, it means that a message arrived and the process should process the message. Otherwise, the process should handle the error.

- If a section contains duplicated keys, GetPrivateProfileSection() returns the duplicated keys, but all values are the same as the value of the first key.

  Suppose a section contains these keys:

  ```
  key1=x1
  key2=x2
  key2=x3
  key2=x4
  key3=x5
  key2=x6
  ```

  The values returned are:

```
      key1=x1
      key2=x2
      key2=x2
      key2=x2
      key3=x5
      key2=x2
```

- String IDs of resources should not be longer than 255 characters.

- String IDs must be in the English language. The resources themselves can be mulitilingual.

- GetDlgItemInt() only translates up to 16-bit int/unsigned values. This is because it gets its value from Windows, which translates only 16-bit values. As a workaround, call GetDlgItem() and translate the value using atoi() or sscanf().

Additional reference words: limits
KBCategory: kbprg
KBSubcategory: W32s

WinDbg Message "Breakpoint Not Instantiated"

PSS ID Number: Q99953

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

The WinDbg message "Breakpoint Not Instantiated" indicates that the debugger could not resolve an address. This can happen for several reasons:

- A specified symbol does not exist. In this case, check for misspelling and check the state of the "ignore case" option if the symbol contains mixed case.

  -or-

- The symbol exists, but the EXE or DLL was built with the wrong debugging information, or none at all. Use the -Zi and -Od compiler options and use the -debug:full, -debugtype:cv, and -pdb:none linker options.

  -or-

- The symbol exists, but it is in a module that has not yet been loaded. If the symbol is in a DLL that is dynamically loaded the breakpoint was probably set before the DLL was loaded. The message is harmless, because WinDbg will instantiated the BP when the module is loaded.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbtool
KBSubcategory: TlsWindbg

Window Message Priorities

PSS ID Number: Q96006

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

Under Windows NT, messages have the same priorities as with Windows 3.1.

With normal use of GetMessage() (passing zeros for all arguments except the LPMSG parameter) or PeekMessage(), any message on the application queue is processed before user input messages. And input messages are processed before WM_TIMER and WM_PAINT "messages."

MORE INFORMATION
================

For example, PostMessage() puts a message in the application queue. However, when the user moves the mouse or presses a key, these messages are placed on another queue (the system queue in Windows 3.1; a private, per-thread input queue in Win32).

GetMessage() and its siblings do not look at the user input queue until the application queue is empty. Also, the WM_TIMER and WM_PAINT messages are not handled until there are no other messages (for the thread) to process. The WM_TIMER and WM_PAINT messages can be thought of as boolean toggles, because multiple WM_PAINT or WM_TIMER messages waiting in the queue will be combined into one message. This reduces the number of times a window must repaint itself.

Under this scheme, prioritization can be considered tri-level. All posted messages are higher priority than user input messages because they reside in different queues. And all user input messages are higher priority than WM_PAINT and WM_TIMER messages.

The only difference in the Windows NT model from the Windows versions 3.x model is that there is effectively a system queue per thread (for user input messages) rather than one global system queue. The prioritization scheme for messages is identical.

For information concerning SendMessage() from one thread to another, please see the following article(s) in the Microsoft Knowledge Base:

   ARTICLE-ID: Q95000
   TITLE   : SendMessage() in a Multithreaded Environment

Additional reference words: 3.10 3.50 3.51 4.00 95

KBCategory: kbui
KBSubcategory: UsrWndw

Window Owners and Parents

PSS ID Number: Q84190

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

In the Windows environment, two relationships that can exist between windows are the owner-owned relationship and the parent-child relationship.

The owner-owned relationship determines which other windows are automatically destroyed when a window is destroyed. When window A is destroyed, Windows automatically destroys all of the windows owned by A.

The parent-child relationship determines where a window can be drawn on the screen. A child window (that is, a window with a parent) is confined to its parent window's client area.

This article discusses these relationships and some Windows functions that provide owner and parent information for a given window.

MORE INFORMATION
================

Although WS_OVERLAPPED windows don't typically have owners, they can. Please see Kyle Marsh's MSDN article "Win32 Window Hierarchy and Styles," which says:

> Overlapped windows may own other top-level windows or be owned by other top-level windows or both.

Alternatively, the desktop window can be considered the owner and parent of a WS_OVERLAPPED-style window. A WS_OVERLAPPED-style window can be drawn anywhere on the screen and Windows will destroy any existing WS_OVERLAPPED-style windows when it shuts down.

A window created with the WS_POPUP style does not have a parent by default; a WS_POPUP-style window can be drawn anywhere on the screen. A WS_POPUP-style window will have a parent only if it is given one explicitly through a call to the SetParent function.

The owner of a WS_POPUP-style window is set according to the hWndParent parameter specified in the call to CreateWindow that

created the pop-up window. If hWndParent specifies a nonchild window, the hWndParent window becomes the owner of the new pop-up window. Otherwise, the first nonchild ancestor of hWndParent becomes the owner of the new pop-up window. When the owner window is destroyed, Windows automatically destroys the pop up. Note that modal dialog boxes work slightly differently. If hWndParent is a child window, then the owner window is the first nonchild ancestor that does not have an owner (its top-level ancestor).

A window created with the WS_CHILD style does not have an explicit owner; it is implicitly owned by its parent. A child window's parent is the window specified as the hWndParent parameter in the call to CreateWindow that created the child. A child window can be drawn only within its parent's client area, and is destroyed along with its parent.

An application can change a window's parent by calling the SetParent function after the window is created. Windows does not provide a method to change a window's owner.

Windows provides three functions that can be used to find a window's owner or parent:

- GetWindow(hWnd, GW_OWNER)
- GetParent(hWnd)
- GetWindowWord(hWnd, GWW_HWNDPARENT)

GetWindow(hWnd, GW_OWNER) always returns a window's owner. For child windows, this function call returns NULL. Because the parent of the child window behaves similar to its owner, Windows does not maintain owner information for child windows.

The return value from the GetParent function is more confusing. GetParent returns zero for overlapped windows (windows with neither the WS_CHILD nor the WS_POPUP style). For windows with the WS_CHILD style, GetParent returns the parent window. For windows with the WS_POPUP style, GetParent returns the owner window.

GetWindowWord(hWnd, GWW_HWNDPARENT) returns the window's parent, if it has one; otherwise, it returns the window's owner.

Two examples of how Windows uses different windows for the parent and the owner to good effect are the list boxes in drop-down combo boxes and the title windows for iconic MDI (multiple document interface) child windows.

Due to its size, the list box component of a drop-down combo box may need to extend beyond the client area of the combo box's parent window. Windows creates the list box as a child of the desktop window (hWndParent is NULL); therefore, the list box will be clipped only by the size of the screen. The list box is owned by the first nonchild ancestor of the combo box. When that ancestor is destroyed, the list box is automatically destroyed as well.

When an MDI child window is minimized, Windows creates two windows: an

icon and the icon title. The parent of the icon title window is set to the MDI client window, which confines the icon title to the MDI client area. The owner of the icon title is set to the MDI child window. Therefore, the icon title is automatically destroyed with the MDI child window.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

WindowFromPoint() Caveats

PSS ID Number: Q65882

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

SUMMARY
=======

When the coordinates passed to the WindowFromPoint() function correspond to a disabled, hidden, or transparent child window, the handle of that window's parent is returned.

To retrieve the handle of a disabled, hidden, or transparent child window, given a point, the ChildWindowFromPoint() function must be used.

MORE INFORMATION
================

The following code fragment demonstrates the use of the ChildWindowFromPoint() function during the processing of a WM_MOUSEMOVE message. This code finds the topmost child window at a given point, regardless of the current state of the window.

In this fragment, hWnd is the window receiving this message and is assumed to have captured the mouse via the SetCapture() function.

```
HWND  hWndChild, hWndPoint;
POINT pt;
    .
    .
    .
    case WM_MOUSEMOVE:
        pt.x = LOWORD(lParam);
        pt.y = HIWORD(lParam);

        /*
         * Convert point to screen coordinates. When the mouse is
         * captured, mouse coordinates are given in the client
         * coordinates of the window with the capture.
         */
        ClientToScreen(hWnd, &pt);

        /*
         * Get the handle of the window at this point. If the window
         * is a control that is disabled, hidden, or transparent, then
         * the parent's handle is returned.
```

```
         */
        hWndPoint = WindowFromPoint(pt);

        if (hWndPoint == NULL)
            break;

        /*
         * To look at the child windows of hWnd, screen coordinates
         * need to be converted to client coordinates.
         */
        ScreenToClient (hWndPoint, &pt);

        /*
         * Search through all child windows at this point. This
         * will continue until no child windows remain.
         */
        while (TRUE)
            {
            hWndChild = ChildWindowFromPoint(hWndPoint, pt);

            if (hWndChild && hWndChild != hWndPoint)
                hWndPoint = hWndChild;
            else
                break;
            }

        // Do whatever processing is desired on hWndPoint

        break;
```

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrWndw

Windows 95 Does Not Support Template Files

PSS ID Number: Q138955

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API)
  included with Microsoft Windows 95 version 4.0

SUMMARY
=======

The hTemplate parameter in the CreateFile API is not supported in Windows 95. It is supported only in Windows NT.

MORE INFORMATION
================

The hTemplate parameter in CreateFile allows you to create a new file using another file as a template. The template is used to set the file attributes (normal, hidden, system, and read-only) of the new file. It is also used to copy the extended attributes (EA) from the template to the new file.

Additional reference words: 4.00
KBCategory: kbprg
KBSubcategory: BseFileio

Windows 95 Network Programming Support

PSS ID Number: Q125702

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

Windows 95 supports the network programming interfaces listed in the table below:

Interface	Comments
Microsoft RPC	Supports a subset of the protocol sequences specified in Microsoft RPC 1.0. For additional information, please see the following article(s) in the Microsoft Knowledge Base: ARTICLE-ID:Q125701 TITLE :Windows 95 RPC: Supported Protocol Sequences
Windows Sockets (WinSock)	Supports the TCP/IP and IPX/SPX network transports. For additional information, please see the following article(s) in the Microsoft Knowledge Base: ARTICLE-ID:Q125704 TITLE :Multiprotocol Support for Windows Sockets
NetBIOS	Supports NetBIOS v3.0, including the Microsoft extension function NCBENUM.
Network DDE (NetDDE)	Includes the NetDDE agent and a 16 bit NetDDE API DLL. 32-bit applications must thunk to NDDEAPI.DLL. For additional information, please see the following article(s) in the Microsoft Knowledge Base: ARTICLE-ID:Q125703 TITLE :Windows 95 Support for Network DDE

Additional reference words: 4.00
KBCategory: kbnetwork kbnetwork
KBSubcategory: NtwkMisc

Windows 95 RegOpenKeyEx Incompatible with Windows NT

PSS ID Number: Q137202

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

SYMPTOMS
========

A Win32-based application runs under Windows 95 but does not run correctly under Windows NT. Or a Windows-based application runs under Windows 95 but leaks registry handles under Windows NT.

CAUSE
=====

Windows NT RegOpenKeyEx always returns a unique handle. Windows 95 RegOpenKeyEx returns the same handle each time the same key is opened. Windows 95 keeps a reference count, incrementing it each time the key is opened with RegOpenKeyEx and decrementing it each time the key is closed with RegCloseKey. The key remains open as long as the reference count is greater than zero.

Consider the following code:

```
RegOpenKeyEx(OldKey, NULL, 0, MAXIMUM_ALLOWED, &NewKey)
RegCloseKey(NewKey)
RegQueryValue(NewKey,...)
RegCloseKey(NewKey)
```

This code is incorrect, but it works under Windows 95 because OldKey and NewKey refer to the same key. However, the code fails under Windows NT, because NewKey is not valid after it is closed.

In a related issue, RegOpenKey with a NULL subkey string on Windows NT will return the same handle that was passed in. Under Windows 95, the reference count is incremented.

```
RegOpenKey(OldKey, NULL, 0, MAXIMUM_ALLOWED, &NewKey)
RegCloseKey(OldKey)
RegQueryValue(NewKey,...)
RegCloseKey(NewKey)
```

This code works under Windows 95 because NewKey is still valid after closing OldKey, but the code fails under Windows NT. Code that is correct for Windows NT (don't close the handle until both OldKey and NewKey are no longer needed) leaks registry handles under Windows 95.

RESOLUTION
==========

You should use RegOpenKeyEx and make sure that there is a corresponding close call for each open call. This will ensure that you do not use any handle that has been closed.

STATUS
======

This behavior is by design.

Additional reference words: 4.00 Windows 95
KBCategory: kbprg kbprb
KBSubcategory: BseMisc

Windows 95 RPC: Supported Protocol Sequences

PSS ID Number: Q125701

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

The Windows 95 RPC runtime libraries support a subset of the protocol sequences introduced in Microsoft RPC 1.0. Here is the list of supported protocol sequences:

Protocol Sequence	Comments
ncacn_ip_tcp	Requires TCP/IP installation
ncacn_nb_nb	NetBEUI is the only supported NetBIOS transport
ncacn_np	Support for client applications only
ncacn_spx	Requires IPX/SPX installation
ncalrpc	Local communication only

Regarding ncacn_np: Windows 95 does not support server-side named pipes, so ncacn_np is not a valid protocol sequence for RPC servers running on Windows 95. However, ncacn_np is valid for RPC client applications.

Additional reference words: 4.00 NETWORKING
KBCategory: kbnetwork
KBSubcategory: NtwkRpc

Windows 95 Support for LAN Manager APIs

PSS ID Number: Q125700

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK)

Windows 95 supports a subset of the Microsoft LAN Manager API. This is a list of the supported functions:

```
NetAccessAdd
NetAccessCheck
NetAccessDel
NetAccessEnum
NetAccessGetInfo
NetAccessGetUserPerms
NetAccessSetInfo
NetConnectionEnum
NetFileClose2
NetFileEnum
NetSecurityGetInfo
NetServerGetInfo
NetServerSetInfo
NetSessionDel
NetSessionEnum
NetSessionGetInfo
NetShareAdd
NetShareDel
NetShareEnum
NetShareGetInfo
NetShareSetInfo
```

Windows 95 support for these functions differs from Windows NT in two ways. First, since Windows 95 doesn't support Unicode, these functions require ANSI strings. Second, Windows 95 exports the Lan Manager functions from SVRAPI.DLL instead of NETAPI32.DLL. If an attempt is made to run a native Windows NT application on Windows 95, the following error will result:

> "The <application> file is linked to missing export NETAPI32.DLL
> <Net...API>"

To handle these differences, applications targeted to both Windows NT and Windows 95 should do the following:

1. Avoid importing Lan Manager functions from NETAPI32.DLL at link time. Instead, applications should do a run time version check and dynamically link to NETAPI32.DLL for Windows NT or SVRAPI.DLL for Windows 95.

   For additional information on version checking, please see the following article(s) in the Microsoft Knowledge Base:

ARTICLE-ID:Q92395
TITLE :Determining System Version from a Win32-based Application

2. Make sure the application doesn't depend on the presence of unsupported API's.

3. When calling Lan Manager API's, pass strings using a character set appropriate for the host operating system. Use Unicode strings for Windows NT and ANSI strings for Windows 95.

If you are only targetting Windows 95 and wish to use SVRAPI.DLL, SVRAPI.H and SVRAPI.LIB are included in the Windows 95 DDK. NOTE: The formal parameter lists for the LAN Manager APIs may be slightly different between the header files for Windows NT and Windows 95.

Additional reference words: 4.00 95
KBCategory: kbnetwork
KBSubcategory: NtwkLmapi

Windows 95 Support for Network DDE

PSS ID Number: Q125703

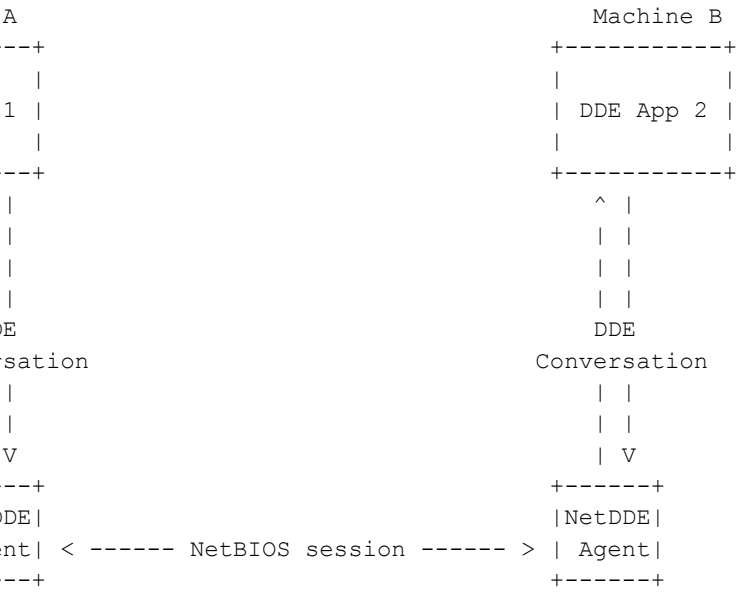
The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows 95 version 4.0

Network DDE is a technology that allows applications that use the DDE transport to transparently exchange data over a network. It consists of two major components:

1. The NetDDE agent. This is a service that acts as a proxy for the remote DDE application. It communicates with all local DDE applications, and with remote NetDDE agents using NetBIOS as shown in this chart:



2. A DLL that implements NetDDE API functions such as NDdeShareAdd, NDdeShareDel, and so on. This DLL is usually named NDDEAPI.DLL.

In the interests of backwards compatibility, Windows 95 includes a NetDDE agent and a 16-bit NetDDE API DLL. However, Windows 95 does not include a 32-bit NetDDE API DLL. Consequently, 32-bit applications that use NetDDE API functions will need to thunk to the 16-bit NETAPI.DLL.

Additional reference words: 4.00
KBCategory: kbnetwork kbui
KBSubcategory: UsrNetDde

Windows 95 Uses Known16DLLs Registry Key to Find 16-bit DLLs

PSS ID Number: Q141969

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  Microsoft Windows 95

SUMMARY

The process of loading a DLL, implicitly or explicitly, invokes Windows 95 to search for the specified DLL in a predefined order until the DLL is found or all search efforts are exhausted. In Windows 95, the string values associated with the Known16DLLs registry key have the ability to reset the normal search order used to locate and load a 16-bit DLL to a new predefined order. Thus, a string value that identifies a 16-bit DLL and is also associated with the Known16DLLs registry key will force Windows 95 to begin its search for the DLL in the System directory - not the current directory.

MORE INFORMARTION

In Windows 95, the predefined order that is used to locate and load a 16-bit DLL is specified as follows:

1. The current directory.

2. The Windows directory (the directory containing Win.com).

3. The Windows system directory (the directory containing such system files as Gdi.exe).

4. The directory containing the executable file for the current task.

5. The directories listed in the PATH environment variable.

This search order is reset if a 16-bit DLL being loaded has a string value with the same name as the 16-bit DLL specified in the Known16DLLs registry key. The Known16DLLs registry key is located at:

   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionManager

The data associated with the DLL string value can take any form. An example of such a string value is:

Name	Data
"COMMCTRL.DLL"	"SYSTEM COMMCTRL.DLL"

When a string value for a 16-bit DLL exists, the new search order is as

follows:

1. The Windows system directory.

2. The Windows directory.

3. The current directory.

4. The directory from which the application loaded.

5. The directories that are listed in the PATH environment variable.

There are three ways that this DLL string value can be added to the registry:

- An end user can use Regedit.exe to add the 16-bit DLL string value to the Known16DLLs registry key.

  -or-

- The 16-bit DLL string value can be added programmatically to the Known16DLLs registry key using registry APIs.

  -or-

- Windows 95 can automatically add the 16-bit DLL string value to the Known16DLLs registry key.

The first two ways to add the 16-bit DLL string value are basic and require no further explanation; however, the third method is more complex. Each time a 16-bit DLL is loaded from the Windows System directory, Windows 95 checks the Known16DLLs registry key for a string value that has that same name. If the string value does not exist, Windows 95 creates it in the Known16DLLs registry key. Thus, any application that uses a DLL that has the same name as a DLL listed in the Known16DLLs registry key will always use the reset search order to locate the DLL. As long as the DLL exists in the Windows System directory and can be loaded by another application, deletion of the string value will not be effective in returning the search order to its original predefined state.

Additional reference words: search path 16 3.1 loadlibrary win95 order
KBCategory: kbenv kbhowto
KBSubcategory: BseDll

Windows 95 Uses KnownDLLs Registry Key to Find 32-bit DLLs

PSS ID Number: Q151646

The information in this article applies to:

- Microsoft Win32 Application Programming Interface included with:
  - Microsoft Windows 95

- Microsoft Win32 Software Development Kit for:
  - Microsoft Windows 95

SUMMARY
=======

The process of loading a DLL, implicitly or explicitly, invokes Windows 95 to search for the specified DLL in a predefined order until the DLL is found or all search efforts are exhausted. In Windows 95, the string values associated with the KnownDLLs Registry key have the ability to reset the normal search order used to locate and load a 32-bit DLL to a new predefined order. Thus, a string value that identifies a 32-bit DLL and is also associated with the KnownDLLs Registry key will force Windows 95 to begin its search for the DLL in the System directory, not the current directory.

MORE INFORMATION
================

In Windows 95, the predefined order used to locate and load a 32-bit DLL is specified as follows:

1. The directory from which the application loaded.

2. The current directory.

3. The Windows system directory. Use the GetSystemDirectory function to get the path of this directory.

4. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.

5. The directories that are listed in the PATH environment variable.

This search order is reset if:

1. The DLL name passed to LoadLibrary specifies the .DLL extension.

2. The 32-bit DLL specified has a Registry string value with the same name, excluding the extension, as the 32-bit DLL specified in the KnownDLLs Registry key. The KnownDLLs Registry key is located at:

   HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionManager

The data associated with the DLL string value must specify the actual DLL

name for the DLL to be found and loaded. This DLL name does not need to match the string value name and is not required to following any naming convention. Some examples of such string values are:

Name	Data
MYDLL1	MYDLL.DLL
MYDLL2	MYREALDLL2.DLL

When a string value for a 32-bit DLL exists, the new search order in the Windows system directory is as follows:

1. If MYDLL1.DLL is passed to LoadLibrary, the Registry key MYDLL1 is used to load MYDLL.DLL.

2. If the DLL name specified in the data value, MYDLL.DLL for example, cannot be found, LoadLibrary on MYDLL1.DLL will fail and a call to GetLastError will return error 2 "The system cannot find the file specified".

3. If MYDLL.DLL or MYDLL is passed to LoadLibrary, the normal search pattern is used to locate and load MYDLL.DLL.

Following are ways that this DLL string value can be added to the Registry:

- An end user can use Regedit.exe to add the 32-bit DLL string value to the KnownDLLs Registry key.

  -or-

- The 32-bit DLL string value can be added programmatically to the Known32DLLs Registry key using Registry APIs.

To revert back to the default search pattern, the Registry key must be removed.

Additional reference words: search path loadlibrary win95 order
KBCategory: kbenv kbusage kbprg
KBSubcategory: BseDll

Windows Dialog-Box Style DS_ABSALIGN

PSS ID Number: Q11590

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

The Windows dialog-box style DS_ABSALIGN (used in WINDOWS.H) means "Dialog Style ABSolute ALIGN." Specifying this style in the dialog template tells Windows that the dtX and dtY values of the DLGTEMPLATE struct are relative to the screen origin, not the owner of the dialog box. When this style bit is not set, the dtX and dtY fields are relative to the origin of the parent window's client area.

Use this term if the dialog box must always start in a specific part of the display, no matter where the parent window is on the screen.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDlgs

Windows Does Not Support Nested MDI Client Windows

PSS ID Number: Q74041

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY
=======

The Microsoft Windows implementation of the multiple document interface (MDI) does not support nested MDI client windows. In other words, neither an MDI client window nor an MDI child window can have an MDI client window as a child.

MORE INFORMATION
================

A Windows MDI client window is a member of the MDIClient window class, and the Windows MDI model assumes that the parent of an MDIClient window is a top-level frame window with a valid menu bar. This assumption is necessary to implement the basic functionality defined by the MDI interface, and it precludes the possibility of using nested MDIClient windows. However, an application can have multiple top-level windows, and each top-level window can have a separate MDIClient window as a child.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMdi

Windows Help PositionWindow Macro Documented Incorrectly

PSS ID Number: Q83911

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) version 3.1
- Microsoft Win32 SDK, versions 3.5 and 3.51

SUMMARY

In source files for Windows Help, the help author can specify the size, position, and state of a window with the PositionWindow macro.

MORE INFORMATION

According to page 326 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 4: Resources," the state parameter to the PositionWindow macro is either 0 to specify a normal sized window or 1 to specify a maximized window. This information is incorrect. The state parameter can assume any of ten different values, as follows:

Value	Corresponding Windows Constant	Action
0	SW_HIDE	Hides the window and passes activation to another window.
1	SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position.
2	SW_SHOWMINIMIZED	Activates the window and displays it as an icon.
3	SW_SHOWMAXIMIZED	Activates the window and displays it as a maximized window.
4	SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The window that is currently active remains active.
5	SW_SHOW	Activates a window and displays it in its current size and position.
6	SW_MINIMIZE	Minimizes the specified window and activates the top-level window in the window-manager's list.

7	SW_SHOWMINNOACTIVE	Displays the window as iconic. The window that is currently active remains active.
8	SW_SHOW	Displays the window in its current state. The window that is currently active remains active.
9	SW_RESTORE	Same as SW_SHOWNORMAL.

Note that these constants are valid only for Windows and may change if Help is ported to another platform.

Additional reference words: 3.10 3.50
KBCategory: kbtool
KBSubcategory: TlsHlp

Windows NT Servers in Locked Closets

PSS ID Number: Q90083

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY

=======

Some installations are required to restrict access to a server such that access to the server's keyboard/mouse is unavailable to most personnel. Such a server is referred to as a server in a locked closet.

The server administrators may provide an emergency reset button to end users (for example, factory floor workers) in case the system locks up and no administrators are present. In the case where an emergency reset button cannot be provided, an administrator must come and physically unlock the closet in order to get the system up again.

Windows NT requires that the user press CTRL+ALT+DEL to log on. This requirement implies that Windows NT doesn't lend itself well to the server in a locked closet situation. Indeed, someone must press CTRL+ALT+DEL and enter a user ID and a password in order to log on and use the keyboard or mouse to interact locally with a Windows NT machine. However, it is possible to configure the machine as a server in a locked closet such that an administrator is not required to unlock the door to reset the system. The administrator can configure the system so that services are started automatically during boot. Once all the services are started, then the system is fully functional and the administrator need not intervene. If certain services fail to come up, but network service does come up, then the system can be remotely administered.

MORE INFORMATION

================

Remote administration is possible, assuming that the required basic system services are running. The machine must be on the network. The process requires only the Windows NT product. In other words, Windows NT Advanced Server is not an additional requirement.

Make sure that the following steps have been taken to start system services automatically at system boot and to enable remote administration in case of failure:

1. Use the Service Control Manager to install any application code that must be started as soon as the Windows NT machine reboots.

   Write an application that installs the services and specifies that they should be started automatically. To find more information on the Win32 APIs that support Services, search on "Services Overview" in the Win32

API help text.

Once this is done, the necessary application code can be made to start automatically upon system reboot, without anyone needing to press CTRL+ALT+DEL to log on or to take any other action using the server's local mouse/keyboard.

2. Make sure that the Workstation and Server services start automatically upon reboot.

   Use the Services application in Control Panel to ensure that both the Workstation and Server services start automatically upon reboot. Choose the Help button for instructions on how to install a service and how to configure it to start automatically.

   This will permit an authorized person to remotely administer the system from another machine on the network. Thus, if something from step 1 goes wrong, the administrator still does not need to physically unlock the closet and log on. The administrator can log on to any machine on the network and use the tools on that machine to interact with the server.

Remote administration via dial-up telephone lines is available, but requires RAS (Microsoft Remote Access Service). RAS permits a machine to dial over telephone lines into a network, and to become a full participant on the network. In this way, a system dialing in over RAS can be used to remotely administer the system in the locked closet.

Note that while these steps allow servers locked in closets to be restored without an administrator, it is still preferable to install a UPS (uninterruptable power supply). Servers in locked closets usually need to provide uninterrupted service to their clients, so a UPS is a better solution. The capability to do remote administration serves as a backup in case of failure.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMisc

Windows NT Support for the MS-DOS LAN Manager APIs

PSS ID Number: Q110776

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT versions 3.1 and 3.5

There is no list that shows specifically which MS-DOS LAN Manager APIs (application programming interfaces) are supported in a VDM (virtual DOS machine) on Windows NT. This article discusses what influenced whether or not an API would be implemented.

The set of APIs that is supported is relatively small. It was necessary to implement named pipes, mailslots, NetServerEnum(), and the NetUseXXX APIs. The APIs that are commonly used in shipping applications were implemented, if possible. There were certain APIs that were impossible to implement from the VDM. The remaining APIs were not added either because Microsoft did not feel that they were used in applications or because they did not make sense in this context, such as the NetServiceXXX APIs.

Additional reference words: 3.10 3.50
KBCategory: kbnetwork
KBSubcategory: NtwkLmapi

Windows NT Virtual Memory Manager Uses FIFO

PSS ID Number: Q98216

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1 and 3.5

On page 193 of "Inside NT," Helen Custer states that the Windows NT virtual memory manager uses a FIFO (first in, first out) algorithm, as opposed to a LRU (least recently used) algorithm, which the Windows virtual memory manager uses. While it is true that FIFO can result in a commonly used page being discarded or paged to the pagefile, there are reasons why this algorithm is preferable.

Here are some of the advantages:

- FIFO is done on a per-process basis; so at worst, a process that causes a lot of page faults will slow only itself down, not the entire system.

- LRU creates significant overhead--the system must update its page database every single time a page is touched. However, the database may not be properly updated in certain circumstances. For example, suppose that a program has good locality of reference and uses a page constantly so that it is always in memory. The operating system will not keep updating the timestamp in the page database, because the process is not hitting the page table. Therefore this page may age even though it is in nearly constant use.

- Pages that are "discarded" are actually kept in memory for a while, so if a page is really used frequently, it will be brought back into memory before it is written to disk.

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMm

Windows Regions Do Not Scale

PSS ID Number: Q71229

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

The coordinates of Windows regions are in device units, not logical units. Therefore, if the mapping mode is not MM_TEXT, region coordinates do not scale as would be expected. This causes particular problems in metafiles because metafiles often use MM_ISOTROPIC or MM_ANISOTROPIC modes to make pictures appear the same on devices with different resolutions.

To work around this problem, applications should avoid using regions if the mapping mode is changed from MM_TEXT. Regions should also be avoided in metafiles, unless the application knows the scaling factor and can adjust region coordinates itself.

MORE INFORMATION

If the applications that will read and write the metafile are developed together, the writing application can include an MFCOMMENT escape in the metafile to store the region components. This information can be used by the reading application to scale the metafile.

However, this comment is not standard across all applications. This method would not be expected to work if the metafile is imported into a commercial application.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiDc

Windows Socket API Specification Version

PSS ID Number: Q85965

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK), versions 3.1, 3.5, 3.51, and 4.0

The Windows Sockets API Specification version 1.1 is now available in two formats. WINSOCK.DOC is a Word for Windows, version 2.0, document and WINSOCK.TXT is an ASCII-text-format document.

These files contain the Windows Sockets API Specification version 1.1, which defines a standard binary interface for tcp/ip transports based on the Berkeley Sockets interface originally in Berkeley UNIX with Windows-specific extensions. This specification has been endorsed by 20 leading companies, and defines the sockets interface in Windows NT. The specification will be supported by various vendors in their upcoming tcp/ip product releases for Windows for MS-DOS.

The Windows Sockets API Specification is contained in a file called WINSOCK (contains ASCII text version) and a file called WINSOCKW (contains a Word for Windows version) which is located in the Microsoft Software Library.

Download WINSOCK.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download WINSOCK.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \softlib\mslfiles directory
  Get WINSOCK.EXE

Download WINSOCKW.EXE, a self-extracting file, from the Microsoft Software Library (MSL) on the following services:

- Microsoft Download Service (MSDL)
  Dial (206) 936-6735 to connect to MSDL
  Download WINSOCKW.EXE

- Internet (anonymous FTP)
  ftp ftp.microsoft.com
  Change to the \softlib\mslfiles directory
  Get WINSOCKW.EXE

Additional reference words: 1.00 1.10 3.10 3.50 4.00 95
KBCategory: kbref
KBSubcategory: NtwkWinsock

Windows WM_SYSTIMER Message Is an Undocumented Message

PSS ID Number: Q108938

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The WM_SYSTIMER message in Windows is an undocumented system message; it should not be trapped or relied on by an application. This message can occasionally be viewed in Spy or in CodeView while debugging.

Windows uses the WM_SYSTIMER message internally to control the scroll rate of highlighted text (text selected by the user) in edit controls, or highlighted items in list boxes.

NOTE: The WM_SYSTIMER message is for Windows's internal use only and can be changed without prior notice.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrMsg

WM_ACTIVATEAPP Has lParam of Zero (0)

PSS ID Number: Q135785

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY
=======

Although the current Win32 documentation does not specifically indicate this, it is possible (and very likely) that the WM_ACTIVATEAPP message will be received by an application with the lParam equal to 0. You can in fact get 0 in lParam when activating and deactivating.

MORE INFORMATION
================

In previous versions of Windows, activation and deactivation used to be synchronous. This meant that you could never switch away from an application that was locked up because the application becoming active had to wait for a synchronous reply from the application being deactivated. The activating application would get stuck waiting for the deactivating application to process and return from the WM_ACTIVATEAPP, WM_ACTIVATE, and WM_NCACTIVATE messages.

In Windows 95 and Windows NT, Microsoft removed every place where these types of synchronous lockups can occur. When asynchronous activation occurs, the activating application becomes active immediately, and the messages to the deactivating application occur later. This means two things:

- An application can be terminated in the middle of this process. If this happens, the thread ID/task handle that is passed in the lParam is invalid.

- When the deactivated application gets notified, it is possible that the activated application is no longer active. A third application might have been activated in the interim.

Additional reference words: 4.00 1.30 Windows 95
KBCategory: kbui
KBSubcategory: UsrWndw

WM_CHARTOITEM Messages Not Received by Parent of List Box

PSS ID Number: Q72552

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

When keyboard input is sent to a list box that has the LBS_WANTKEYBOARDINPUT style bit set, its parent does not receive WM_CHARTOITEM messages; however, WM_VKEYTOITEM messages are received. This is because the list box has the LBS_HASSTRINGS style bit set.

This behavior is by design. Windows sets the LBS_HASSTRINGS style bit for all list boxes except owner-draw list boxes. An owner-draw list box can be created with this style bit turned on or off. For owner-draw list boxes, the state of the LBS_HASSTRINGS style bit determines which messages are sent. WM_CHARTOITEM messages and WM_VKEYTOITEM messages are mutually exclusive.

The documentation for WM_CHARTOITEM states:

> Only owner-draw list boxes that do not have the LBS_HASSTRINGS style can receive this message.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95 listbox
KBCategory: kbui
KBSubcategory: UsrInp

WM_COMMNOTIFY is Obsolete for Win32-Based Applications

PSS ID Number: Q94561

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

Under Windows version 3.1, the WM_COMMNOTIFY message is posted by a communication device driver whenever a COM port event occurs. The message indicates the status of a window's input or output queue.

This message is not supported for Win32-based applications. However, WOW supports the EnableCommNotification() API for 16-bit Windows-based applications running on Windows NT.

MORE INFORMATION

To duplicate the Windows 3.1 functionality for a Win32-based application, refer to the TTY sample, included with the SDK. The TTY sample is a common code base sample, which uses EnableCommNotification() under Windows 3.1 to tell COMM.DRV to post messages to the TTY window.

In Win32, this behavior is simulated with a secondary thread which uses WaitCommEvent() to block on the port and PostMessage() to indicate when the desired event has occurred.

TTY.C defines WM_COMMNOTIFY if WIN32 is defined. Using this method, WM_COMMNOTIFY notifications are simulated but use the same message definition as Windows 3.1.

The TTY sample is located on the Win32 SDK CD in \MSTOOLS\SAMPLES\COMM.

Additional reference words: 3.10 3.50 4.00 95
KBCategory: kbprg
KBSubcategory: BseCommapi

WM_CTLCOLORxxx Message Changes for Windows 95

PSS ID Number: Q130952

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows 95 version 4.0

SUMMARY
=======

In previous versions of Windows, an application could change the background, text, and/or text background colors of controls by performing certain actions in response to WM_CTLCOLORxxx messages.

However, in Windows 95, the messages sent by different types of controls are somewhat different.

MORE INFORMATION
================

The following list outlines the changes in WM_CTLCOLORxxx messages sent by standard controls in Windows 95:

WM_CTLCOLORBTN

Sent By: command buttons (regular and default)

Changes made during this message have no effect on command buttons. Command buttons always use system colors for drawing themselves.

WM_CTLCOLORSTATIC

Sent By: Any control that displays text which would be displayed using the default dialog/window background color. This includes check boxes, radio buttons, group boxes, static text, read-only or disabled edit controls, and disabled combo boxes (all styles).

The changes affect the text drawn in the control. Changes do not affect the checkmarks on the buttons or the outline of the group box.

WM_CTLCOLOREDIT

Sent By: Enabled, non-read-only edit controls and enabled combo boxes (all styles)

The changes affect the background, text, and text background of these controls. For combo boxes, the changes made in this message affect only the "edit" portion of the control. The list portion is affected by the

WM_CTLCOLORLISTBOX message.

In previous versions of Windows, radio buttons, check boxes and group boxes would send WM_CTLCOLORBTN messages and paint themselves accordingly. In Windows 95, these controls send WM_CTLCOLORSTATIC messages instead.

These changes were implemented to make changing the appearance of controls more logical (text on the dialog background is now classified as "static").

Additional reference words: 4.00
KBCategory: kbui
KBSubcategory: UsrCtl

WM_DDE_EXECUTE Message Must Be Posted to a Window

PSS ID Number: Q77842

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

Chapter 15 of the "Microsoft Windows Software Development Kit Reference, Volume 2" documents the dynamic data exchange (DDE) protocol. The following statement is found on page 15-2:

> An application calls the SendMessage() function to issue the WM_DDE_INITIATE message or a WM_DDE_ACK message sent in response to WM_DDE_INITIATE. All other messages are sent using the PostMessage() function.

In the book "Windows 3: A Developer's Guide" by Jeffrey M. Richter (M & T Computer Books), the sample setup program uses the SendMessage() function to send itself a WM_DDE_EXECUTE message that violates the DDE protocol and may not work in future versions of Windows.

In Richter's sample, no real DDE conversation exists. The correct method to achieve the desired result is to use the SendMessage() function to send a user-defined message to the window procedure. When this message is processed, proceed accordingly.

For more information on user-defined messages, see chapter 6 of the "Microsoft Windows Software Development Kit Reference, Volume 1" for the Windows SDK version 3.0 and chapter 2 of the "Programmer's Reference, Volume 3: Messages, Structures, and Macros" from the Windows SDK version 3.1.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

WM_KEYDOWN Under Japanese Windows 95 and Windows NT

PSS ID Number: Q150021

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK), version 3.1
- Microsoft Win32 Software Development Kit (SDK) for:
  - Microsoft Windows NT version 3.51
  - Microsoft Windows 95

On Microsoft Japanese Windows 95, the IME system no longer uses the keyboard hook to pick up keyboard messages. Therefore, all keyboard messages are passed on to the application. The result is that the same application running under Japanese Windows 95 receives WM_KEYDOWN messages and WM_KEYUP messages, while an application running under Japanese Windows 3.1. does not receive WM_KEYDOWN messages.

If you do not want your application running under Japanese Windows 95 to receive WM_KEYDOWN messages while the IME is working, you can modify the Win.ini file as follows:

```
[IME Compatibility]
APPNAME=0x00010000
```

where APPNAME is the name of the application.

Additional reference words: 4.00 3.50 3.51 IME
KBCategory: kbprg
KBSubcategory: wintldev

WM_SIZECLIPBOARD Must Be Sent by Clipboard Viewer App

PSS ID Number: Q74274

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

    - Microsoft Windows NT versions 3.5 and 3.51
    - Microsoft Windows 95 version 4.0

The WM_SIZECLIPBOARD message is not generated by the Windows graphical environment; the clipboard viewer generates this message.

An application that can display the clipboard contents, such as the CLIPBRD.EXE application in Windows version 3.0, is a clipboard viewer. When the size of the clipboard viewer's client area changes and the clipboard contains a data handle for the CF_OWNERDISPLAY format, the clipboard viewer must send the WM_SIZECLIPBOARD message to the current clipboard owner. The GetClipboardOwner function returns the window handle of the current clipboard owner. This window handle is the handle in the last call to OpenClipboard.

When sending the WM_SIZECLIPBOARD message, the clipboard viewer must specify two parameters. The wParam parameter identifies the clipboard viewer's window handle. The low-order word of the lParam parameter contains the handle to a block of global memory that holds a RECT data structure. The RECT structure defines the area in the clipboard viewer that the clipboard owner should paint.

Additional reference words: 3.00 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrClp

WM_SYSCOLORCHANGE Must Be Sent to Windows 95 Common Controls

PSS ID Number: Q129595

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
    - Microsoft Windows NT version 3.51
    - Microsoft Windows 95 version 4.0
    - Microsoft Win32s version 1.3

SUMMARY

=======

You must ensure that applications that use the new common controls introduced in Windows 95 and Windows NT version 3.51 forward the WM_SYSCOLORCHANGE message to the controls.

MORE INFORMATION

================

Windows 95 makes it very easy for the user to change the colors of common user interface objects, therefore it is critical that applications not rely on particular colors being constant. When the user changes the color settings, Windows 95 will send a WM_SYSCOLORCHANGE message to all top level windows. Because this message is sent only to top level windows, the common controls will not be notified of the color change unless the application forwards the WM_SYSCOLORCHANGE message to the control.

An example of why this is important is the toolbar control. If the color settings are such that the "3D Objects" color is set to light gray, the toolbar will create its buttons to light gray. However if the WM_SYSCOLORCHANGE message is not forwarded to the toolbar and the 3D Object color is changed to blue, the toolbar buttons will remain light gray while all the other buttons in the system change to blue.

Additional reference words: 1.30 4.00 grey

KBCategory: kbui

KBSubcategory: UsrCtl

Working Set Size, Nonpaged Pool, and VirtualLock()

PSS ID Number: Q108449

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1 and 3.5

SUMMARY
=======

This article discusses memory management issues such as working set size, nonpaged pool, and locking pages into physical memory via VirtualLock().

MORE INFORMATION
================

Working Set

A process's working set is the set of pages that it has currently in memory. The values for maximum working set and minimum working set are hard-coded in Windows NT and are thus impossible to change. There are three values hard-coded for the maximum working set. The value used for maximum working set depends on whether your machine is considered to be a small, medium, or large machine:

   Small machine--less than 16 megabytes (MB)
   Medium machine--between 16 MB and 20 MB
   Large machine--greater than 20 MB

The system tries to keep about 4 MB free to reduce the paging that occurs when loading a new process, allocating memory, and so forth. Any free RAM beyond what the system requires is available for the system to use to increase your working set size if your process is doing a lot of paging.

Process pages that are paged out of your process space are moved into the "standby list," where they remain until sufficient free RAM is available, or until system memory is low and they need to be reused. If these pages are accessed by your process while they are still on the standby list and more RAM has become available, they will be "soft-faulted" back into the working set. This does not require any disk access, so it is very quick. Therefore, even though you have an upper limit to the size of your working set, you can still have quite a few process pages in memory that can be pulled back into your working set very quickly.

To minimize working set requirements and increase performance, use the Working Set Tuner, WST.EXE, to order the functions within your code. One way to help an application receive a larger working set is to use the Network application in the Control Panel to set the server configuration to "Maximize Throughput for Network Applications."

Nonpaged Pool

System memory is divided into paged pool and nonpaged pool. Paged pool can be paged to disk, whereas nonpaged pool is never paged to disk. In Windows NT 3.1, the default amount of nonpaged pool also depends on whether your machine is considered small, medium, or large. In other words, you will have X amount of nonpaged pool on a 16 MB machine, Y amount of nonpaged pool on a 20 MB machine, and Z amount of nonpaged pool on a machine with more than 20 MB (the exact values for X, Y, and Z were not made public).

Important system data is stored in nonpaged pool. For example, each Windows NT object created requires a block of nonpaged pool. In fact, it is the availability of nonpaged pool that determines how many processes, threads, and other such objects can be created. The error that you will receive if you have too many object handles open is:

```
1816 (ERROR_NOT_ENOUGH_QUOTA)
```

Many 3.1 applications ran into this error because of the limited amount of nonpaged pool. This limit were addressed in Windows NT 3.5. We found that

- Some objects were too large
- Sharing an object caused excessive quota charges
- The quota limits were artificial and fixed

The resources used by each object were evaluated and many were drastically reduced in Windows NT 3.5.

In Windows NT 3.1, every time an object was shared, quota was charged for each shared instance. For example, if you opened a file inheritable and then spawn a process and have it inherit your handle table, the quota charged for the file object was double. Each handle pointing to an object cost quota. Most applications experienced this problem. Under Windows NT 3.5, quota is only charged once per object rather than once per handle.

Windows NT 3.1 had a fixed quota for paged and nonpaged pool. This was determined by the system's memory size, or could be controlled by the registry. The limits were artificial. This was due to the poor design of quotas with respect to sharing. It was also affected by some objects lying about their actual resource usage. In any case, Windows NT 3.5 has revised this scheme.

The Windows NT 3.1 "Resource Kit, Volume I" documents that it is possible to change the amount of nonpaged pool by modifying the following registry entry:

```
HKEY_LOCAL_MACHINE\SYSTEM\
   CurrentControlSet\
   Control\
   Session Manager\
   Memory Management\
   NonPagedPoolSize
```

WARNING: This modification can cause the system to crash, and therefore

Microsoft does not recommend that this registry entry be changed.

Quotas can still be controlled in Windows NT 3.5 using these Windows NT 3.1 registry values. However, this technique is now almost never needed. The new quota mechanism dynamically raises and lowers your quota limits as you bump into the limits. Before raising a limit, it coordinates this with the memory manager to make sure you can safely have your limit raised without using up all of the systems resources.

VirtualLock()

To lock a particular page into memory so that it cannot be swapped out to disk, use VirtualLock(). The documentation for VirtualLock() states the following:

> Locking pages into memory may degrade the performance of the system by reducing the available RAM and forcing the system to swap out other critical pages to the paging file. There is a limit on the number of pages that can be locked: 30 pages. The limit is intentionally small to avoid severe performance degradation.

There is no way to raise this limit in Windows NT 3.1--it is fixed at 30 pages (the size of your working set). The reason that you see a severe performance degradation when an application locks these pages is that Windows NT must reload all locked pages whenever there is a context switch to this application. Windows NT was designed to minimize page swapping, so it is often best to let the system handle swapping itself, unless you are writing a device driver that needs immediate access to memory.

Windows NT 3.5 allows processes to increase their working set size by using SetProcessWorkingSetSize(). This API is also useful to trim your minimum working set size if you want to run many processes at once, because each process has the default minimum working set size reserved, no matter how small the process actually is. The limit of 30 pages does not apply to VirtualLock() when using SetProcessWorkingSetSize().

Additional reference words: 3.10 3.50
KBCategory: kbprg
KBSubcategory: BseMm

Writing Code that Works with Different International Formats

PSS ID Number: Q130056

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
- Microsoft Win32 Software Development Kit (SDK) version 3.5
- Microsoft Win32s version 1.2

SUMMARY

Some European coutries like Germany use a different floating point format that doesn't include the decimal point (.). This has caused problems whtn the actual numbers use the decimal point format (xxx.xx) and the German version of Microsoft Excel is expecting a comma (xxx,xx).

Some developers have made the mistake of hard-coding formats or searching for all periods and replacing them with commas. This is not good programming practice.

Microsoft Excel uses the Control Panel Number Format for its separators. So, regardless of what language version of Windows it runs on, it always uses the correct separators. Microsoft recommends that you use the same approach. Always use the separators defined in the number format of control panel.

MORE INFORMATION

All the international information is stored in the WIN.INI file under the [intl] heading. The application should look for the value of "sDecimal." It can query this value using the GetProfileString API. If the application uses this approach, it will have the correct separator for all countries.

To avoid potential problems, never make assumptions about number formats, currency formats (that is, don't hard-code the dollar symbol), date formats, or time formats. These are different for different countries. Use the settings from the Control Panel.

If an application does any text processing, such as sorting or upper/lowercase conversions, it should use the Windows APIs and not supply its own conversion tables and functions. For example, if an application use APIs such as AnsiLower, it will work regardless of language because it obtains the data from the language DLL.

To output floating point numbers in Windows, Microsoft recommends that you use wsprintf. The concept of an "international format string" only has meaning when the string is output to the user. Internal to the computer, for all calculations and manipulations, the concept does not apply. Thus the application can keep its floats in the native format and only convert to strings before calling the wsprintf function.

There is no need to load conversions from float to string, and vice versa. In other words, there is no need to write any functions such as InterStringToFloat(), InterFloatToString(), and so on.

Here's a simple algorithm you could use.

1. Convert your float to a string by using the _fcvt or _gcvt standard C functions.

2. Get the sThousand and sDecimal separators from the WIN.INI file by using the GetProfileString API.

3. Call wsprintf to output the converted string, using sThousand and sDecimal in the formatting string.

The following is an example of a mistake made by a developer who was not thinking internationally:.

```
   if language=English
      Output_English_Float ()
   else
      Output_International_Float ()
```

This is not correct. Think of English as just another language and write a single Output_Float() function that covers all languages.

Also, for ease of localization, a developer should place all strings in the resource file. This avoids having to search through all the C files looking for strings to translate.

Additional reference words: 1.20 3.10 3.50 localization kbinf foreign
KBCategory: kbother
KBSubcategory: wintldev

Writing Multiple-Language Resources

PSS ID Number: Q89866

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

When you are writing multiple-language resources, the dialog box identifiers need to be identical for each language instance, as demonstrated below.

```
#define DialogID       100

DialogID DIALOG  0, 0, 210, 10
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US
.
.
.
DialogID DIALOG  0, 0, 210, 10
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH
```

The FindResource() application programming interface (API) function is used by the system to fetch the dialog box. FindResource() gets the locale information for the process, then attempts to fetch the resource with that language identifier using FindResourceEx(), the language-specific API function for fetching resources. If FindResourceEx() fails to load the language-specific dialog box, FindResource() then attempts to load the neutral dialog box, which should fetch LANG_FRENCH,SUBLANG_FRENCH, if the locale is SUBLANG_FRENCH_CAN or similar.

The LANGUAGE identifiers and the VERSIONINFO language identifiers should also be identical. The code page for resources is always the Unicode code page. The system will translate from Unicode to the required code page.

The preferred method of developing multiple-language resources is to include a LANGUAGE statement for each language supported rather than using the CODEPAGE, LANGUAGE identifier, and VERSIONINFO information. Although the CODEPAGE information will work, the new method is easier to use.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrNls WintlDev

WS_EX_WINDOWEDGE Doesn't Work Without a Window Frame Style

PSS ID Number: Q136311

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows 95 version 4.0

The WS_EX_WINDOWEDGE style is new to Windows 95. The style specifies that a window has a border with a raised edge. If the WS_EX_WINDOWEDGE style is used on a window that doesn't have either the WS_THICKFRAME or WS_DLGFRAME style, the WS_EX_WINDOWEDGE style has no effect. This is because the WS_EX_WINDOWEDGE style modifies the window's frame. Therefore, if no frame is specified, then the window does not receive the 3-D effect.

This problem typically occurs when you are trying to give a raised edge to a control. Because controls by default have no frame, only a border, the WS_EX_WINDOWEDGE style is ignored. To work around the problem, add the WS_DLGFRAME style to the control.

Additional reference words: 95 raised edge child window.
KBCategory: kbui
KBSubcategory: UsrWndw

wsprintf() Buffer Limit in Windows

PSS ID Number: Q77255

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.1 and 3.0

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.1, 3.5, and 3.51
  - Microsoft Windows 95 version 4.0

The wsprintf(lpOutput, lpFormat [, argument] ...) and wvsprintf() functions format and store a series of characters and values in a buffer specified by the first parameter, lpOutput. This buffer is limited to 1K (1024 bytes); in other words, the largest buffer that wsprintf can use is 1K.

If an application tries to use a buffer larger than 1K, the string will be truncated automatically to a length of 1K.

Additional reference words: 3.00 3.10 3.50 4.00 95
KBCategory: kbgraphic
KBSubcategory: GdiDrw

XFONT.C from SAMPLES\OPENGL\BOOK Subdirectory

PSS ID Number: Q124870

The information in this article applies to:

- Microsoft Win32 Application Programming Interface (API) included with:
  - Microsoft Windows NT version 3.5

SUMMARY

XFONT.C from the SAMPLES\OPENGL\BOOK subdirectory is not in the MAKEFILE, and subsequently is never built.

MORE INFORMATION

XFONT.C should not be built because the sample uses X Windows system-specific functions. To use fonts with Windows NT version 3.5 OpenGL, use the wglUseFontBitmaps() and glCallLists() functions as described in the OpenGL online reference. Another alternative is to use auxDrawStr(). Note that auxDrawStr is defined in GLAUX.H as:

```
#ifdef UNICODE
#define auxDrawStr auxDrawStrW
#else
#define auxDrawStr auxDrawStrA
#endif
void APIENTRY auxDrawStrA(LPCSTR);
void APIENTRY auxDrawStrW(LPCWSTR);
```

REFERENCES

For further information on using fonts with OpenGL, please refer to the OpenGL online reference titled "Fonts and Text."

Additional reference words: 3.50
KBCategory: kbgraphic kbgraphic
KBSubcategory: GdiOpenGL

XTYP_EXECUTE and its Return Value Limitations

PSS ID Number: Q102574

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.1

- Microsoft Win32 Application Programming Interface (API) included with:

  - Microsoft Windows NT versions 3.5 and 3.51
  - Microsoft Windows 95 version 4.0

SUMMARY

A DDEML client application can use the XTYP_EXECUTE transaction to cause a server to execute a command or a series of commands. To do this, the client creates a buffer that contains the command string, and passes either a pointer to this buffer or a data handle identifying the buffer, to the DdeClientTransaction() call.

If the server application generates data as a result of executing the command it received from the client, the return value from the DdeClientTransaction() call does not provide a means for the client to access this data.

MORE INFORMATION

For an XTYP_EXECUTE transaction, the DdeClientTransaction() function returns TRUE to indicate success, or FALSE to indicate failure. In most cases, this provides inadequate information to the client regarding the actual result of the XTYP_EXECUTE command.

Likewise, the functionality that DDEML was supposed to provide through the lpuResult parameter of the DdeClientTransaction() function upon return is currently not supported, and may not be supported in future versions of DDEML. The lpuResult parameter was initially designed to provide the client application access to the server's actual return value (for example, DDE_FACK if it processed the execute, DDE_FBUSY if it was too busy to process the execute, or DDE_FNOTPROCESSED if it denied the execute).

In cases where the server application generates data as a result of an execute command, the client has no means to get to that data, nor does it have a means to determine the status of that execute command through the DdeClientTransaction() call.

An example of this might be one where the DDEML client application specifies a command to a server application such as "OpenFile <FileName>" to open a file, or "DIR C:\WINDOWS" to get a list of files in a given directory.

There are two ways that the client application can work around this limitation and gain access to the data generated from the XTYP_EXECUTE command:

Method 1

The client can issue an XTYP_REQUEST transaction (with the item name set to "ExecuteResult", for example) immediately after its XTYP_EXECUTE transaction call returns successfully. The server can then return a data handle in response to this request, out of the data generated from executing the command.

Method 2

The client can establish an ADVISE loop with the server (with topic!item name appropriately set to Execute!Result, for example) just before issuing the XTYP_EXECUTE transaction. As soon as the server then executes the command, it can immediately update the advise link by calling DdePostAdvise(), and return a data handle out of the data generated from executing the command. The client then receives the data handle in its callback function, as an XTYP_ADVDATA transaction.

Note that these workarounds apply only if one has access to the server application's code. Third-party server applications that provide no means to modify their code as described above can't obtain any data generated by the application as a result of an XTYP_EXECUTE back to the client.

Additional reference words: 3.10 3.50 3.51 4.00 95
KBCategory: kbui
KBSubcategory: UsrDde

Rebasing Win32 DLLs: The Whole Story

Ruediger R. Asche
Microsoft Developer Network Technology Group

September 18, 1995

Abstract

This article discusses the ramifications of dynamic-link library (DLL) rebasing under both Microsoft® Windows NT® and Windows® 95. ("Rebasing" in this context refers to the process of changing the base address of a DLL in memory space.) A sample application accompanies this article, as well as a suite of DLLs to provide comparison figures.

Introduction

One of the questions I have heard a lot recently from developers at Microsoft is, "Gee, what happens if the operating system has to rebase my DLLs? What is the penalty for that, and is there any way that I can prevent the penalty? Is there any way I can change the code to generate fewer fixups?"

I thought that was a really good question, so I decided to temporarily relocate to Empiric-land, investigate the costs of DLL loading, and pour a bucket of numbers at your feet so that you can decide for yourself what to do about the DLLs.

The results presented in this paper are probably not revolutionary, nor are they surprising: Prefer one large DLL over several small ones; make sure that the operating system does not need to search for the DLLs very long; and avoid many fixups if there is a chance that the DLL may be rebased by the operating system (or, alternatively, try to select your base addresses such that rebasing is unlikely). However, as the old saying goes, "The journey is the goal." In other words, on the way to writing this paper, I found a number of little things about DLLs and memory management that I think are worth sharing. A more appropriate title for the paper might actually have been "Bits and Pieces about DLLs."

In this paper I describe a sample test application that I wrote to measure DLL loading times, as well as providing a set of DLLs to measure.

The Application

The architecture of the test set to measure dynamic-link library (DLL) load times is very simple: The PAGETEST application, written using the Microsoft® Foundation Class libraries (MFC), consists of two threads. The first (main application) thread creates and owns a mutex object. This first thread samples the current time and then calls **LoadLibrary** to explicitly load one of a number of libraries I provide (I discuss the libraries in the next section). Meanwhile, the second thread waits for the mutex object to become signaled.

All of the libraries consist of the DLL entry procedure only. In the PROCESS_ATTACH dispatch point of the DLL entry procedure, the mutex object is signaled. At that point, the secondary application thread wakes up and computes the difference between the current time and the time sampled before **LoadLibrary** was called. This difference is roughly the elapsed time that was used to load the DLL into memory. The MFC application has an option to load and unload the DLL repeatedly (50 times) so that a meaningful average loading time can be computed.

I will not discuss the specifics of the application here—it's a fairly straightforward MFC application, with all the relevant code located in the view class. The view is derived from **CEasyOutputView** to provide for easy display of results. (Please see "Windows NT Security in Theory and Practice" for details.)

Note that this empirical test has a number of drawbacks that may distort the actual results:

- We are assuming that the time-sampling mechanism is efficient, reliable, and has a granularity that is fine enough. (I use the system performance counters.)
- We are assuming that the thread-switching mechanism is consistently efficient and does not have too bad an effect on the time it takes to wake up the secondary thread.
- The results from the tests naturally depend to a high degree on the underlying hardware (that is, the processor speed of the machine on which the test is run, the number of processors, the speed of the hard disk controller, and so forth).
- The results were sampled using specific versions of the software involved (operating system versions, C run-time library versions, and so forth).
- Normally most DLLs are loaded implicitly, not explicitly, and it is assumed that the implicit and explicit loading of DLLs takes the same amount of time, all other things being equal.

To make things worse, the numbers I did obtain vary widely at times.

Thus, you should take the results of the tests with a grain of salt. The most important deduction to make from the results is not the absolute load times, but the relative times—in other words, how changing one property changes the loading behavior, and how different strategies compare to each other.

If you wish to recreate the test results on your machine, follow the DLL positioning instructions in the next section, run PTAPP.EXE, and choose Run All Tests from the Multiple Test menu.

The DLLs

I considered a number of properties of DLLs relevant to their load time:

- The size of the library
- The number of items to be relocated
- Whether the DLL initializes C startup code or not
- Whether the DLL exports symbols or not
- Whether the DLL implicitly links to other libraries or not
- How long it takes the operating system to locate the DLL executable

Aside from these issues, there are also a few factors independent of the DLL that determine how slowly or quickly a DLL can be loaded—for example, the underlying operating system, the overall current work load on the machine, the application's working set, whether the DLL needs to be rebased, and so forth.

To make a long story short, I wrote 18 little (or not-so-little) DLLs that represent almost all permutations of the following properties:

- DLL size (can be small or large)
    - If it's a large DLL, has load-time fixups or not
- C run-time support (none, implicitly linked, or explicitly linked)
- Symbols exported from the DLL (yes or no)

I loaded each of the 18 DLLs, under both Windows NT® version 3.51 and Windows® 95 on the same machine, at its preferred base address and with the preferred virtual memory range taken. Each test was also run first with the DLL located in the current directory and then deep down in the path to measure how long it takes the operating system to locate the DLL in the search path. As I mentioned earlier, each test was run 50 times to obtain a meaningful average value.

The first observation I made was that under Windows NT, the initial load time for any given DLL was about three times the time it would subsequently take to load the same DLL on the average. This is a side effect of Windows NT's memory management design: Once the DLL is initially loaded and subsequently unloaded, the pages that belong to the DLL image remain in memory; they are inserted into what is called the standby list (a system-maintained list of discarded pages that can be made available to the application if it should need the pages again or if another application requesting new memory should need them). For a more thorough description of the standby list, please consult Helen Custer's *Inside Windows NT*, pages 194-196.

Reloading the DLL's pages from the standby pages is much more efficient than reloading from the disk. Over time, the pages will migrate from the standby list to the free list such that, if there is a lot of memory allocation and access activity goes on between the initial and subsequent DLL loading tests, the time difference will even out. To simulate this behavior (and make sure that I could obtain meaningful average DLL loading times from several tests), I added a little option that allows the test application to hog as

much memory as it possibly can so that the standby list will be exhausted quickly. There is also a little utility that comes with the Windows NT Resource Kit that can be used to force pages off the standby list (CLEARMEM.EXE).

That worked, but unfortunately, right after I freed the hogged memory, the load time was about 20 times the average load time—or 7 times the initial load time!

This phenomenon put me into some kind of Catch-22 situation: On the one hand, I wanted to obtain a reliable average figure for DLL loading times under normal working conditions; on the other hand, the only reliable and consistent figures I could obtain were not the ones under normal working conditions! My way out of that dilemma is a little daring but, I hope, valid: I base my results on the comparisons between the average DLL load times and assume that the relationships between the initial and subsequent average load times are constant so that the comparison values are still meaningful under normal working conditions.

If you wish to rebuild the DLLs or add your own DLL variations, or if you are just curious to see what I did to build 18 DLLs, read on; otherwise, skip this subsection and continue under the heading titled "The Theory."

Building the DLLs

The DLLs were built using Visual C++™ version 2.2 using a makefile generated by Visual C++. You will find the project in the attached sample code in the PAGETEST subdirectory. Each of the 18 DLLs is built from the same project; you should build each DLL as the retail (no debug) version and then copy the generated executable to a new location using the naming convention that follows.

The PTAPP sample application expects the name of the DLL to encode the information about what the DLL contains. Each letter in the DLL's name represents one property, according to the following scheme:

- The first letter expresses whether the DLL is small (that is, does not contain any data) or large (that is, has 100,000 static data elements). An *S* in this position indicates small, an *L* indicates large, and *F* indicates that all of the 100,000 data elements are initialized to a relocatable string, the address of which must be fixed up at load time when the DLL is rebased.

  Note that 100,000 relocatable strings does not necessarily mean 100,000 relocations. There is a problem with the linker in Visual C++ version 2.*x* that will limit to 64K the number of relocatable items in a portable executable (PE) file. Thus, if you run an .EXE header utility such as YAHU on one of the DLLs whose name begins with an *F*, you will find that there are only about 34K of relocations. This problem will be fixed in upcoming versions of Visual C++.

- The second letter indicates whether the DLL supports C run-time code or not. This letter can either be *N* (meaning that the DLL has a custom entry point that does not call the C run-time initialization code), *C* (meaning that the DLL's entry point is **DllMain**, which implicitly initializes the C run-time support) or *D* (meaning that the DLL has a custom entry point that calls _CRT_INIT to initialize the C run-time library dynamically).

- Finally, the third letter is *N* if the DLL does not export any symbols or *E* if a function is exported. The remaining letters are currently unassigned.

  For example, SCNNNNNN.DLL is a small DLL that implicitly calls the C run-time initialization code, but does not export a symbol. FNENNNNN.DLL is a large DLL with many relocations that does not call the C run-time initialization code but exports a symbol.

In order not to introduce any unwanted side effects into the comparisons, I made the DLLs as small as I possibly could. The smallest DLL I provide has nothing but a custom DLL entry point that does not initialize the C run-time support code.

There is no MFC support in any of the DLLs because MFC DLLs implicitly link to other DLLs and perform custom initializations that I did not want introduced into the measurements. All of the other variations of DLLs are built with small modifications to the project, as follows:

- In order to make a small DLL into a large DLL, add the symbol MANYPAGES to the preprocessor directives. Generate a large DLL with many fixups by adding both MANYPAGES and FIXUPS to the preprocessor definitions. To build a DLL with a custom entry point that calls the C run-time initialization code, add the symbol DYNACRT; to build a DLL with implicit C run-time initialization code, add the symbol STANDARDENTRY and rename the DLL entry point in the Settings/Link/Entry text box to DllMain. Finally, to have the DLL export a symbol, add the preprocessor directive HASSYMBOLS.

- For the ambitious, I also define a symbol, HUGEBINARY, that will generate a really big DLL (a DLL with about 15,000 data pages which, if combined with the FIXUPS symbol, will yield about 15,000

relocations). The DLL is 40 to 61 MB in size, depending on whether you define FIXUPS or not; therefore, I do not include the binary in the DLL test set.

Whatever options you use to build the DLL, the resulting executable will be called PAGETEST.DLL in the WINREL subdirectory of the PAGETEST project. After building the DLL, you should copy the DLL to a different location, renaming the DLL according to the above naming conventions.

To see how searching for the DLL binary affects the load time, I kept two copies of each DLL on my machine—one in the same directory as PTAPP.EXE (the test application) and one in the subdirectory that is listed at the very end in the search path (in my case, C:\DOS). After having run the test with the DLL found in the same directory as the executable, I renamed all of the DLLs to force the operating system to look for the DLLs in another directory.

When I built the DLLs, I ran into a few scenarios where I changed one option for test purposes and was unable to recreate the original configuration afterwards. Thus, just to make sure that you can rebuild the DLLs exactly as I built them, here are the project options I used.

- Compiler

```
/nologo /MT /W3 /GX /YX /O2 /D <see above> /FAcs /Fa "WinRel/" FR
"WinRel/" /Fp

"WinRel/pagetest.pch" /Fo "WinRel" /c
```

- Preprocessor

  The exact preprocessor options depend on the type of library built, as explained before.

- Linker

```
kernel32 advapi msvcrt /nologo /subsystem:windows /DLL /incremental:no
/PDB:

"WinRel/pagetest.pdb" /MACHINE:I386
```

Note The PE file format contains time stamps. That means that if you build the same DLL two times, the resulting binary images will not be identical. A byte-byte-byte file-comparison utility should report six differing bytes in two groups of three consecutive bytes, one for every pair of independently built, but otherwise identical, DLLs.

The Theory

The operating system has to go through these steps to load a DLL:

1. Locate the DLL executable file on disk.

2. Traverse the list of DLLs loaded into the application's address space to determine if the DLL is already loaded.

3. Allocate the memory for the DLL to reside in and map the DLL binary into that memory. (In Windows NT, this happens through section objects.)

4. Perform various manipulations in order for the DLL to work (that is, resolve fixups in the DLL, and so forth).

Various factors determine how fast a DLL will be loaded. Here is a (possibly incomplete) list of the ones that need to be taken into consideration:

- The underlying hardware and software: How fast the computer is and what operating system it runs.

- The current state of the system and the application: How tight the system is on virtual memory, and if the DLL can be loaded at the preferred base address.

- The DLL itself: How big it is, how many locations in the DLL need to be fixed up (in conjunction with two above), and whether the DLL implicitly links to other DLLs that also need to be loaded.

This list tells us that rebasing a DLL is by no means the only factor that determines a DLL's loading time. In this article I present a lot of numbers that should give you an idea of how widely the loading time for a DLL can vary and how much an application can influence the loading time.

Note that rebasing a DLL may result not only in a greater load time, but also in a penalty in pagefile usage. One of the first steps in loading a DLL consists of creating a section object—that is, a contiguous region of memory that is backed by the DLL executable file. Whenever a page of the DLL is removed from an application's working set, the operating system will reload that page from the DLL executable file the next time the page is accessed.

Of course, when a DLL is rebased, this scheme no longer works because the pages that contain relocated addresses differ from the corresponding pages in the DLL executable image. Thus, as soon as the operating system attempts to fix up an address when loading an executable file, the corresponding page is copied (because the section was opened with the COPY_ON_WRITE flag), all the changes are made to the copy, and the operating system makes a note that from now on the page is to be swapped from and to the system pagefile instead of the executable image.

There are two potential performance hits in this setup: First, each page that contains an address to be relocated takes up a page on the system pagefile (which will, in effect, reduce the amount of virtual memory available to all applications); and second, as the operating system performs the first fixup in a DLL's page, a new page must be allocated from the pagefile, and the entire page is copied.

The act of performing fixups also increases a DLL's load time, although the algorithm that scans the relocation section of the DLL and applies the fixups is fairly efficient. (The complexity of the traversal is simply a linear function of the number of fixups to be performed.)

Fixups

A couple of frequently asked questions about DLL rebasing are, "What exactly is a fixup, and is there any way that I can code so that I avoid a lot of fixups in my executable?" The answer to both questions depends to a high degree on the platform for which a particular executable has been built. In this article, I will limit the discussion to executables built for Intel 386, 486, and Pentium processors. (Note that executables built for other platforms have different notions of what a fixup is.)

On 386, 486, or Pentium processors, there are basically two things that can cause an address to be marked as relocatable: static objects and absolute jumps.

First, if a static object is referenced by DLL code, the absolute address of the object is used (assuming that the DLL is loaded into its preferred address). For example, in the code fragment

```
LPSTR lpName="Name";
```

the DLL loader will allocate the string "Name" in the DLL's data segment and fill the beginning address of that string into the location that corresponds to the variable *lpName*. If the string "Name" must be relocated because the DLL could not be loaded at its base address, *lpName* must be updated accordingly. Note that in this case, every reference to *lpName* from within the code must also be fixed up.

Objects that can be subject to relocation are literal strings (for example, the string "Name" in the example above), as well as global and static data of every type, including statically allocated C++ objects. Note that especially in C++ there may be many hidden cross-references from one static object to another. Uninitialized data will (trivially) not be fixed up during the relocation process, but references to uninitialized static data will.

The second category of items that can be relocated in an i386 executable is absolute jumps and function calls, including calls to system functions. Note that there is not much you can do in your code to avoid relocations, except for cutting down on statically allocated data. One way to accomplish that would be to avoid resource references by name in favor of referencing resources by ordinal (inasmuch as each name that you explicitly use in your code automatically becomes a potentially relocatable item).

I would not recommend, however, that you design your DLL code with the specific goal of minimizing load time unless (1) the number of statically allocated objects can be significantly reduced, and (2) such a coding practice does not sacrifice other goals in your software design.

One optimization you can perform rather easily, however, is to sort your relocatable data out into only a few pages. It is obvious that two pages with one relocatable item each will both need to be backed by the pagefile if the DLL needs to be rebased. If both relocatable items will occur in the same page, there is only one page that is affected. You might want to check with the pragma (data_seg) directive to ensure that as many relocatable items as possible go into as few pages as absolutely necessary.

The Tools

The fun part about gathering the DLL load times was that I got to understand the internal workings of the operating systems, as well as the executable format, a little bit better. Here are a few tools I considered very useful for dissecting the DLLs as images and at run time:

- An executable header utility (such as YAHU, written about in the technical article "YAHU, or Yet Another Header Utility") can tell you almost everything about the DLL after it has been built but before it is loaded—for example, how many sections there are, how big each section is, how many relocations there are, and so forth.

- A process walker (for example, PWALK from the Win32 SDK) can tell you where in a process's address space a DLL has been loaded, and where the individual sections of the DLL go.

- A process viewer (for example PVIEW from the Windows NT Resource Kit) tells you how many pages the DLL uses in memory at any given time.

- The performance monitor that is shipped with Windows NT (by default installed in the Administrative Tools group) is an excellent tool to monitor the impact of DLL loading on the system, for example, in terms of pagefile usage.

Let us look at how we can use these tools to get a better understanding of the internal workings of a DLL. Running YAHU on the DLL SNNNNNNN.DLL, we obtain the following information on the five sections in the DLL:

- Section .TEXT, size 0x28 bytes, located 0x400 bytes behind the beginning of the file. This section contains the complete DLL code. This section will be loaded one page (0x1000) behind the start address of the DLL image.

- Section .DATA, size 9, located 0x600 bytes behind the beginning of the file. This section contains all the initialized data and will be loaded two pages (0x2000) behind the start address of the DLL image.

- Section .IDATA, size 0x6c, located 0x800 bytes behind the beginning of the file. In this section you will find the import data; that is, the names of the DLLs that the executable links to and the names of the functions called in those DLLs. This section will be loaded three pages (0x3000) behind the start address of the DLL image.

- Section .EDATA, size 0x35, located 0xA00 bytes behind the beginning of the file. This section contains the DLL's export information and will be loaded four pages (0x4000) behind the start address of the DLL image.

- Section .RELOC, size 0x32, located 0xC00 bytes into the file. This section contains the complete relocation information for the entire DLL and will be loaded five pages (0x5000) behind the start of the DLL image in memory.

In other DLLs, you may find more sections—for example, the .BSS section, which contains uninitialized data.

Note that the offsets of the respective sections in the file help you to look at the binary data. For example, open the DLL in binary mode in Visual C++, and scroll down to offset 0xc00. You will see eight bytes of

heading followed by six data bytes. The exact format of the relocation records is described in the *Microsoft Systems Journal* article "Peering Inside the PE: A Tour of the Win32 Executable File Format" (Pietrek 1994) in the Development Library. Note that the information in the .RELOC section gives you all you need to determine where in memory the relocations will be performed.

Thus, the DLL image of SNNNNNNN.DLL consists of six pages: The PE header and the five sections listed above, each of which happens to consist of one page. Now run PTAPP.EXE under control of PWALK, and select a small DLL with no exports and no CRT support from the Select DLL menu. You should see a message saying that SNNNNNNN.DLL was located somewhere on your hard drive. Choose Load DLL from the Run Single Tests menu. You should now see a message saying that the DLL was loaded at some address. Then go back to PWALK, rewalk the process, and scroll down to the address that PTAPP reported as the loading address (if the DLL was loaded at the preferred base address, this would be 0x10000000). You will then see the six pages of the DLL exactly in the order they were specified in the executable header. Note that the page that belongs to the .RELOC section is listed as a second page in the .EDATA section.

Then run PVIEW.EXE and select the process PTAPP.EXE from the process list combo box. In the User Address Space group box, select SNNNNNNN.DLL from the combo box. You should now see all of the DLL's pages sorted by access type: The DLL is listed as occupying a total of 24K (six pages). 12K (or three pages) are listed as *read-only*—the DLL header page beginning at 0x10000000 and the two pages in the .EDATA section beginning at 0x10004000. One page in the .IDATA section is marked as *read/write*. (This must be *read/write* because an import designation may refer to a DLL that must be rebased, so entries in this section may actually have to be updated.) The one page in the .TEXT section is marked as *execute*, and the .DATA section page has *copy-on-write* protection.

If you run the same procedure on one of the large DLLs, you will see that the .DATA section will grow as expected, and all of the relocatable data in that section will be marked as *copy-on-write*. As mentioned before, the *copy-on-write* scheme ensures that relocations will be performed not on the physical page of the DLL image, but on a copy on the pagefile.

The Numbers

One of the caveats I mentioned earlier about measuring DLL load times is that your mileage may vary greatly. I ran the test set several times and found that, although some patterns and general relationships can be detected, the influence of the overall machine work load may skew the results widely—differences of up to 20 percent from one test run to the other are not atypical.

Let me first describe how I obtained the numbers and then interpret the results. Please refer to Appendix A for the test runs on which I based the evaluations in this paper.

In order to obtain a set of numbers, run the test application PTAPP and choose Run All Tests from the Run Multiple Tests menu. This will invoke a script that loads all of the 18 DLLs 50 times each. (An individual scenario can be tested by choosing a particular DLL from the Select DLL menu, choosing Finish to locate the DLL and initialize the test, and then choosing Run Without Hogging from the Run Multiple Tests menu. A DLL can be loaded in a one-shot fashion using the Load DLL menu item from the Run Single Tests menu.) Caution: The test takes several minutes to complete.

The result of each test will be displayed in the application's main window. The first line displays the resolution of the system performance counter (this can be used to compute absolute times), and after the last test, you will find a table of 36 figures. These numbers are the average load times (in performance-counter ticks) for each of the 18 DLLs loaded both at the preferred address and rebased. As I mentioned earlier, the number of ticks, in conjunction with the performance counter resolution, can be used to compute the absolute loading times through this formula:

loading times in seconds = number of ticks/ performance counter resolution.

The test application also computes the relative load time in parentheses behind each result; this is based on the smallest result encountered while running the test.

In order to obtain the four sets of 36 numbers each (as listed in Appendix A), you should run the test application four times: twice under Windows NT (once with the DLLs located in the same directory as PTAPP.EXE, and once with the DLLs located deep down in the search path), and twice under Windows 95 (same conditions).

As I mentioned before, none of the results I present are groundbreakingly new nor surprising. Here are the important conclusions:

- The figures for Windows NT and Windows 95 do not differ wildly, except that Windows 95 does seem to load small DLLs slower and large DLLs faster than Windows NT.

- All other things being equal, the size of the DLL does not matter; that is, the costs for loading a small DLL and a large DLL are pretty much equal. Thus, if possible, you should avoid writing a lot of small DLLs and instead write fewer large DLLs if load time is an issue for you. Note that this observation holds true over a very wide range of DLL sizes—when I ran the test on the huge binary DLL I mentioned earlier (the one with 15,000 pages), the load time did not differ very much from the load time for the small DLL that contains six pages total.

- Rebasing the DLL incurs an overhead of about 600 percent on Windows NT and around 400 percent on Windows 95. Note, however, that this implies a great number of fixups (34,000 in the sample suite). For a typical DLL, the number is much smaller on the average; for example, in the debug version of MFC30D.DLL, which ships with Visual C++ version 2.*x*, there are about 1700 fixups, which is about 5 percent of the 34,000 fixups in the sample suite.

- The single biggest factor that slows down the loading of DLLs is the *location* of the DLL. The

documentation for **LoadLibrary** describes the algorithm that the operating system uses for locating the DLL image; a DLL located at the first search position (the current directory) loads in typically 20 percent or less of the time as the same DLL located deep down in the path loads. It is fairly obvious that the exact load time difference depends a lot on the length of the path, the efficiency of the underlying file system, and the number of files and directories that need to be searched.

Here are the numbers in neat, digestible format. Please refer to Appendix A for information on how the numbers were pulled together.

{ewc msdncd, EWGraphic, bsd23456 0 /a "SDKW.WPG"}

Figure 1. Windows NT 3.51 DLL load times

{ewc msdncd, EWGraphic, bsd23456 1 /a "SDKW.WPG"}

Figure 2. Windows 95 DLL load times

The Recommendations

The single, major thing you can do to speed up DLL loading is to ensure that the operating system does not spend a lot of time locating the DLL—either put the DLL in the same directory from which the executable is started, or start the executable with your environment variable set up so that the DLL in question can be located quickly. This is something you can do without even touching the DLL. If you load the DLL repeatedly and explicitly, you can use the **SearchPath** application programming interface (API) to first obtain the full path name of the DLL location so that you can provide the operating system with an exact location before loading the DLL.

The other main optimization that can help you speed up DLL loading—if there are a significant number of relocation items in the DLL—is to try to ensure that the DLL will not have to be rebased by the operating system. You will also notice that for very small DLLs, the presence of the C run-time initialization code may slow down the DLL loading a little bit.

As you can see from the numbers above, there is a fixed cost in loading a DLL, regardless of its size; thus, you are much better off writing one bigger DLL instead of a number of small DLLs.

Finally, I need to reiterate that due to the way both Windows NT and Windows 95 handle the management of pages that are to be discarded (the pages are, in fact, kept in memory and will be reused over time), the loading of an executable is much faster if the same executable has already been loaded into any application's address space or has recently been loaded and is still on the standby list.

What Else Is There?

I wouldn't like to end this article without mentioning another issue that is related to DLL loading: binding import addresses to external DLLs. Fortunately for me, there is no need to explicitly discuss this issue here because pretty much everything that is to be said has already been said in Matt Pietrek's article series on DLL binding in the "Windows Q&A" column in *Microsoft Systems Journal*, which describes the internals of DLL import binding as well as the usage of the BIND utility. (See the reference in the "Bibliography" section.)

Bibliography

Custer, Helen. *Inside Windows NT*. Redmond, WA: Microsoft Press, 1993.

Pietrek, Matt. "Peering Inside the PE: A Tour of the Win32 Executable File Format. "*Microsoft Systems Journal* 9 (March 1994). (Development Library, Books and Periodicals)

Pietrek, Matt. "Windows Q&A." *Microsoft Systems Journal* 10 (July 1995). (Development Library, Books and Periodicals)

Pietrek, Matt. "Windows Q&A." *Microsoft Systems Journal* 10 (August 1995). (Development Library, Books and Periodicals)

Appendix A. Results from a Test Run

All tests were executed on an i486 machine running at 33 MHz with 24 MB of RAM. Note that the references (1.0 base value) differ from test set to test set. In the charts, the values from the respective second test sets (DLLs located in the search path) have been adjusted relative to the reference value of the first test set.

Table 1. Windows NT 3.51, DLLs Located in Current Directory (Reference: 1.0 == 17.5 ms)

1a. DLLs Loaded at Preferred Address

DLL Type	Small DLL	Large DLL	Large DLL with Fixups
No CRT, no exports	1.0	1.0	1.0
No CRT, exports	1.1	1.0	1.1
DllMain, no exports	1.25	1.22	1.24
DllMain, exports	1.23	1.18	1.21
CRT_INIT, no exports	1.18	1.2	1.15
CRT_INIT, exports	1.2	1.19	1.21

1b. DLLs Rebased

DLL Type	Small DLL	Large DLL	Large DLL with Fixups
No CRT, no exports	1.25	1.23	6.4
No CRT, exports	1.26	1.3	6.38
DllMain, no exports	1.4	1.4	6.5
DllMain, exports	1.29	1.42	6.52
CRT_INIT, no exports	1.4	1.4	6.45
CRT_INIT, exports	1.3	1.3	6.4

Table 2. Windows NT 3.51, DLLs Located in Search Path (Reference: 1.0 == 85.4 ms)

2a. DLLs Loaded at Preferred Address

DLL Type	Small DLL	Large DLL	Large DLL with Fixups
No CRT, no exports	1.0	1.0	1.0

No CRT, exports	1.0	1.0	1.0
DllMain, no exports	1.0	1.0	1.0
DllMain, exports	1.0	1.0	1.0
CRT_INIT, no exports	1.0	1.0	1.1
CRT_INIT, exports	1.0	1.0	1.0

2b. DLLs Rebased

DLL Type	Small DLL	Large DLL	Large DLL with Fixups
No CRT, no exports	1.1	1.0	2.1
No CRT, exports	1.1	1.0	2.1
DllMain, no exports	1.1	1.1	2.1
DllMain, exports	1.0	1.0	2.1
CRT_INIT, no exports	1.1	1.1	2.1
CRT_INIT, exports	1.1	1.0	2.1

Table 3. Windows 95, DLLs Located in Current Directory (Reference: 1.0 == 21.0 ms)

3a. DLLs Loaded at Preferred Address

DLL Type	Small DLL	Large DLL	Large DLL with Fixups
No CRT, no exports	1.0	1.2	1.2
No CRT, exports	1.0	1.2	1.1
DllMain, no exports	1.0	1.2	1.2
DllMain, exports	1.1	1.2	1.2
CRT_INIT, no exports	1.0	1.2	1.1
CRT_INIT, exports	1.0	1.2	1.2

3b. DLLs Rebased

DLL Type	Small DLL	Large DLL	Large DLL with Fixups

No CRT, no exports	1.1	1.2	4.0
No CRT, exports	1.1	1.2	3.8
DllMain, no exports	1.1	1.2	4.0
DllMain, exports	1.1	1.2	4.0
CRT_INIT, no exports	1.1	1.2	4.0
CRT_INIT, exports	1.1	1.2	4.1

Table 4. Windows 95, DLLs Located in Search Path (Reference: 1.0 == 94.7 ms)

4a. DLLs Loaded at Preferred Address

DLL Type	Small DLL	Large DLL	Large DLL w/ Fixups
No CRT, no exports	1.0	1.0	1.1
No CRT, exports	1.0	1.0	1.0
DllMain, no exports	1.0	1.0	1.1
DllMain, exports	1.0	1.0	1.0
CRT_INIT, no exports	1.0	1.0	1.0
CRT_INIT, exports	1.0	1.0	1.1

4b. DLLs Rebased

DLL Type	Small DLL	Large DLL	Large DLL with Fixups
No CRT, no exports	1.0	1.0	1.7
No CRT, exports	1.0	1.1	1.7
DllMain, no exports	1.0	1.1	1.7
DllMain, exports	1.0	1.0	1.7
CRT_INIT, no exports	1.0	1.1	1.7
CRT_INIT, exports	1.0	1.0	1.7

The Portable Executable File Format from Top to Bottom

Randy Kath
Microsoft Developer Network Technology Group

Created: June 12, 1993

Abstract

The Windows NT® version 3.1 operating system introduces a new executable file format called the Portable Executable (PE) file format. The Portable Executable File Format specification, though rather vague, has been made available to the public and is included on the Microsoft Developer Network CD (Specs and Strategy, Specifications, Windows NT File Format Specifications).

Yet this specification alone does not provide enough information to make it easy, or even reasonable, for developers to understand the PE file format. This article is meant to address that problem. In it you'll find a thorough explanation of the entire PE file format, along with descriptions of all the necessary structures and source code examples that demonstrate how to use this information.

All of the source code examples that appear in this article are taken from a dynamic-link library (DLL) called PEFILE.DLL. I wrote this DLL simply for the purpose of getting at the important information contained within a PE file. The DLL and its source code are also included on this CD as part of the PEFile sample application; feel free to use the DLL in your own applications. Also, feel free to take the source code and build on it for any specific purpose you may have. At the end of this article, you'll find a brief list of the functions exported from the PEFILE.DLL and an explanation of how to use them. I think you'll find these functions make understanding the PE file format easier to cope with.

Introduction

The recent addition of the Microsoft® Windows NT® operating system to the family of Windows™ operating systems brought many changes to the development environment and more than a few changes to applications themselves. One of the more significant changes is the introduction of the Portable Executable (PE) file format. The new PE file format draws primarily from the COFF (Common Object File Format) specification that is common to UNIX® operating systems. Yet, to remain compatible with previous versions of the MS-DOS® and Windows operating systems, the PE file format also retains the old familiar MZ header from MS-DOS.

In this article, the PE file format is explained using a top-down approach. This article discusses each of the components of the file as they occur when you traverse the file's contents, starting at the top and working your way down through the file.

Much of the definition of individual file components comes from the file WINNT.H, a file included in the Microsoft Win32™ Software Development Kit (SDK) for Windows NT. In it you will find structure type definitions for each of the file headers and data directories used to represent various components in the file. In other places in the file, WINNT.H lacks sufficient definition of the file structure. In these places, I chose to define my own structures that can be used to access the data from the file. You will find these structures defined in PEFILE.H, a file used to create the PEFILE.DLL. The entire suite of PEFILE.H development files is included in the PEFile sample application.

In addition to the PEFILE.DLL sample code, a separate Win32-based sample application called EXEVIEW.EXE accompanies this article. This sample was created for two purposes: First, I needed a way to be able to test the PEFILE.DLL functions, which in some cases required multiple file views simultaneously—hence the multiple view support. Second, much of the work of figuring out PE file format involved being able to see the data interactively. For example, to understand how the import address name table is structured, I had to view the .idata section header, the import image data directory, the optional header, and the actual .idata section body, all simultaneously. EXEVIEW.EXE is the perfect sample for viewing that information.

Without further ado, let's begin.

Structure of PE Files

The PE file format is organized as a linear stream of data. It begins with an MS-DOS header, a real-mode program stub, and a PE file signature. Immediately following is a PE file header and optional header. Beyond that, all the section headers appear, followed by all of the section bodies. Closing out the file are a few other regions of miscellaneous information, including relocation information, symbol table information, line number information, and string table data. All of this is more easily absorbed by looking at it graphically, as shown in Figure 1.

{ewc msdncd, EWGraphic, bsd23457 0 /a "SDKL.BMP"}

Figure 1. Structure of a Portable Executable file image

Starting with the MS-DOS file header structure, each of the components in the PE file format is discussed below in the order in which it occurs in the file. Much of this discussion is based on sample code that demonstrates how to get to the information in the file. All of the sample code is taken from the file PEFILE.C, the source module for PEFILE.DLL. Each of these examples takes advantage of one of the coolest features of Windows NT, memory-mapped files. Memory-mapped files permit the use of simple pointer dereferencing to access the data contained within the file. Each of the examples uses memory-mapped files for accessing data in PE files.

Note Refer to the section at the end of this article for a discussion on how to use PEFILE.DLL.

MS-DOS/Real-Mode Header

As mentioned above, the first component in the PE file format is the MS-DOS header. The MS-DOS header is not new for the PE file format. It is the same MS-DOS header that has been around since version 2 of the MS-DOS operating system. The main reason for keeping the same structure intact at the beginning of the PE file format is so that, when you attempt to load a file created under Windows version 3.1 or earlier, or MS DOS version 2.0 or later, the operating system can read the file and understand that it is not compatible. In other words, when you attempt to run a Windows NT executable on MS-DOS version 6.0, you get this message: "This program cannot be run in DOS mode." If the MS-DOS header was not included as the first part of the PE file format, the operating system would simply fail the attempt to load the file and offer something completely useless, such as: "The name specified is not recognized as an internal or external command, operable program or batch file."

The MS-DOS header occupies the first 64 bytes of the PE file. A structure representing its content is described below:

WINNT.H

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
    USHORT e_magic;         // Magic number
    USHORT e_cblp;          // Bytes on last page of file
    USHORT e_cp;            // Pages in file
    USHORT e_crlc;          // Relocations
    USHORT e_cparhdr;       // Size of header in paragraphs
    USHORT e_minalloc;      // Minimum extra paragraphs needed
    USHORT e_maxalloc;      // Maximum extra paragraphs needed
    USHORT e_ss;            // Initial (relative) SS value
    USHORT e_sp;            // Initial SP value
    USHORT e_csum;          // Checksum
    USHORT e_ip;            // Initial IP value
    USHORT e_cs;            // Initial (relative) CS value
    USHORT e_lfarlc;        // File address of relocation table
    USHORT e_ovno;          // Overlay number
    USHORT e_res[4];        // Reserved words
    USHORT e_oemid;         // OEM identifier (for e_oeminfo)
    USHORT e_oeminfo;       // OEM information; e_oemid specific
    USHORT e_res2[10];      // Reserved words
    LONG   e_lfanew;        // File address of new exe header
```

```
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

The first field, **e_magic**, is the so-called magic number. This field is used to identify an MS-DOS-compatible file type. All MS-DOS-compatible executable files set this value to 0x54AD, which represents the ASCII characters *MZ*. MS-DOS headers are sometimes referred to as MZ headers for this reason. Many other fields are important to MS-DOS operating systems, but for Windows NT, there is really one more important field in this structure. The final field, **e_lfanew**, is a 4-byte offset into the file where the PE file header is located. It is necessary to use this offset to locate the PE header in the file. For PE files in Windows NT, the PE file header occurs soon after the MS-DOS header with only the real-mode stub program between them.

Real-Mode Stub Program

The real-mode stub program is an actual program run by MS-DOS when the executable is loaded. For an actual MS-DOS executable image file, the application begins executing here. For successive operating systems, including Windows, OS/2®, and Windows NT, an MS-DOS stub program is placed here that runs instead of the actual application. The programs typically do no more than output a line of text, such as: "This program requires Microsoft Windows v3.1 or greater." Of course, whoever creates the application is able to place any stub they like here, meaning you may often see such things as: "You can't run a Windows NT application on OS/2, it's simply not possible."

When building an application for Windows version 3.1, the linker links a default stub program called WINSTUB.EXE into your executable. You can override the default linker behavior by substituting your own valid MS-DOS-based program in place of WINSTUB and indicating this to the linker with the **STUB** module definition statement. Applications developed for Windows NT can do the same thing by using the **-STUB: linker** option when linking the executable file.

PE File Header and Signature

The PE file header is located by indexing the **e_lfanew** field of the MS-DOS header. The **e_lfanew** field simply gives the offset in the file, so add the file's memory-mapped base address to determine the actual memory-mapped address. For example, the following macro is included in the PEFILE.H source file:

PEFILE.H

```
#define NTSIGNATURE(a) ((LPVOID)((BYTE *)a +    \
                        ((PIMAGE_DOS_HEADER)a)->e_lfanew))
```

When manipulating PE file information, I found that there were several locations in the file that I needed to refer to often. Since these locations are merely offsets into the file, it is easier to implement these locations as macros because they provide much better performance than functions do.

Notice that instead of retrieving the offset of the PE file header, this macro retrieves the location of the PE file signature. Starting with Windows and OS/2 executables, .EXE files were given file signatures to specify the intended target operating system. For the PE file format in Windows NT, this signature occurs immediately before the PE file header structure. In versions of Windows and OS/2, the signature is the first word of the file header. Also, for the PE file format, Windows NT uses a DWORD for the signature.

The macro presented above returns the offset of where the file signature appears, regardless of which type of executable file it is. So depending on whether it's a Windows NT file signature or not, the file header exists either after the signature DWORD or at the signature WORD. To resolve this confusion, I wrote the **ImageFileType** function (following), which returns the type of image file:

PEFILE.C

```
DWORD  WINAPI ImageFileType (
    LPVOID    lpFile)
{
    /* DOS file signature comes first. */
    if (*(USHORT *)lpFile == IMAGE_DOS_SIGNATURE)
        {
        /* Determine location of PE File header from
           DOS header. */
        if (LOWORD (*(DWORD *)NTSIGNATURE (lpFile)) ==
                                IMAGE_OS2_SIGNATURE ||
            LOWORD (*(DWORD *)NTSIGNATURE (lpFile)) ==
                                IMAGE_OS2_SIGNATURE_LE)
            return (DWORD)LOWORD(*(DWORD *)NTSIGNATURE (lpFile));

        else if (*(DWORD *)NTSIGNATURE (lpFile) ==
```

```
                                IMAGE_NT_SIGNATURE)

            return IMAGE_NT_SIGNATURE;


        else

            return IMAGE_DOS_SIGNATURE;

        }


    else

        /* unknown file type */

        return 0;

}
```

The code listed above quickly shows how useful the **NTSIGNATURE** macro becomes. The macro makes it easy to compare the different file types and return the appropriate one for a given type of file. The four different file types defined in WINNT.H are:

WINNT.H

```
#define IMAGE_DOS_SIGNATURE             0x5A4D      // MZ

#define IMAGE_OS2_SIGNATURE             0x454E      // NE

#define IMAGE_OS2_SIGNATURE_LE          0x454C      // LE

#define IMAGE_NT_SIGNATURE              0x00004550  // PE00
```

At first it seems curious that Windows executable file types do not appear on this list. But then, after a little investigation, the reason becomes clear: There really is no difference between Windows executables and OS/2 executables other than the operating system version specification. Both operating systems share the same executable file structure.

Turning our attention back to the Windows NT PE file format, we find that once we have the location of the file signature, the PE file follows four bytes later. The next macro identifies the PE file header:

PEFILE.C

```
#define PEFHDROFFSET(a) ((LPVOID)((BYTE *)a +  \

    ((PIMAGE_DOS_HEADER)a)->e_lfanew + SIZE_OF_NT_SIGNATURE))
```

The only difference between this and the previous macro is that this one adds in the constant SIZE_OF_NT_SIGNATURE. Sad to say, this constant is not defined in WINNT.H, but is instead one I defined in PEFILE.H as the size of a DWORD.

Now that we know the location of the PE file header, we can examine the data in the header simply by assigning this location to a structure, as in the following example:

```
PIMAGE_FILE_HEADER   pfh;
```

```
pfh = (PIMAGE_FILE_HEADER)PEFHDROFFSET (lpFile);
```

In this example, *lpFile* represents a pointer to the base of the memory-mapped executable file, and therein lies the convenience of memory-mapped files. No file I/O needs to be performed; simply dereference the pointer *pfh* to access information in the file. The PE file header structure is defined as:

WINNT.H

```
typedef struct _IMAGE_FILE_HEADER {

    USHORT  Machine;

    USHORT  NumberOfSections;

    ULONG   TimeDateStamp;

    ULONG   PointerToSymbolTable;

    ULONG   NumberOfSymbols;

    USHORT  SizeOfOptionalHeader;

    USHORT  Characteristics;

} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;


#define IMAGE_SIZEOF_FILE_HEADER             20
```

Notice that the size of the file header structure is conveniently defined in the include file. This makes it easy to get the size of the structure, but I found it easier to use the **sizeof** function on the structure itself because it does not require me to remember the name of the constant IMAGE_SIZEOF_FILE_HEADER in addition to the **IMAGE_FILE_HEADER** structure name itself. On the other hand, remembering the name of all the structures proved challenging enough, especially since none of these structures is documented anywhere except in the WINNT.H include file.

The information in the PE file is basically high-level information that is used by the system or applications to determine how to treat the file. The first field is used to indicate what type of machine the executable was built for, such as the DEC® Alpha, MIPS R4000, Intel® x86, or some other processor. The system uses this information to quickly determine how to treat the file before going any further into the rest of the file data.

The *Characteristics* field identifies specific characteristics about the file. For example, consider how separate debug files are managed for an executable. It is possible to strip debug information from a PE file and store it in a debug file (.DBG) for use by debuggers. To do this, a debugger needs to know whether to find the debug information in a separate file or not and whether the information has been stripped from the file or not. A debugger could find out by drilling down into the executable file looking for debug information. To save the debugger from having to search the file, a file characteristic that indicates that the file has been stripped (IMAGE_FILE_DEBUG_STRIPPED) was invented. Debuggers can look in the PE file header to quickly determine whether the debug information is present in the file or not.

WINNT.H defines several other flags that indicate file header information much the way the example described above does. I'll leave it as an exercise for the reader to look up the flags to see if any of them are interesting or not. They are located in WINNT.H immediately after the **IMAGE_FILE_HEADER** structure described above.

One other useful entry in the PE file header structure is the *NumberOfSections* field. It turns out that you need to know how many sections—more specifically, how many section headers and section bodies—are

in the file in order to extract the information easily. Each section header and section body is laid out sequentially in the file, so the number of sections is necessary to determine where the section headers and bodies end. The following function extracts the number of sections from the PE file header:

PEFILE.C

```
int   WINAPI NumOfSections (

    LPVOID    lpFile)

{

    /* Number of sections is indicated in file header. */

    return (int)((PIMAGE_FILE_HEADER)

                PEFHDROFFSET (lpFile))->NumberOfSections);

}
```

As you can see, the **PEFHDROFFSET** and the other macros are pretty handy to have around.

PE Optional Header

The next 224 bytes in the executable file make up the PE optional header. Though its name is "optional header," rest assured that this is not an optional entry in PE executable files. A pointer to the optional header is obtained with the **OPTHDROFFSET** macro:

PEFILE.H

```
#define OPTHDROFFSET(a) ((LPVOID)((BYTE *)a                 + \
    ((PIMAGE_DOS_HEADER)a)->e_lfanew + SIZE_OF_NT_SIGNATURE + \
    sizeof (IMAGE_FILE_HEADER)))
```

The optional header contains most of the meaningful information about the executable image, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information, and so forth. The **IMAGE_OPTIONAL_HEADER** structure represents the optional header as follows:

WINNT.H

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    USHORT  Magic;
    UCHAR   MajorLinkerVersion;
    UCHAR   MinorLinkerVersion;
    ULONG   SizeOfCode;
    ULONG   SizeOfInitializedData;
    ULONG   SizeOfUninitializedData;
    ULONG   AddressOfEntryPoint;
    ULONG   BaseOfCode;
    ULONG   BaseOfData;
    //
    // NT additional fields.
    //
    ULONG   ImageBase;
    ULONG   SectionAlignment;
    ULONG   FileAlignment;
```

```
    USHORT  MajorOperatingSystemVersion;
    USHORT  MinorOperatingSystemVersion;
    USHORT  MajorImageVersion;
    USHORT  MinorImageVersion;
    USHORT  MajorSubsystemVersion;
    USHORT  MinorSubsystemVersion;
    ULONG   Reserved1;
    ULONG   SizeOfImage;
    ULONG   SizeOfHeaders;
    ULONG   CheckSum;
    USHORT  Subsystem;
    USHORT  DllCharacteristics;
    ULONG   SizeOfStackReserve;
    ULONG   SizeOfStackCommit;
    ULONG   SizeOfHeapReserve;
    ULONG   SizeOfHeapCommit;
    ULONG   LoaderFlags;
    ULONG   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

As you can see, the list of fields in this structure is rather lengthy. Rather than bore you with descriptions of all of these fields, I'll simply discuss the useful ones—that is, useful in the context of exploring the PE file format.

Standard Fields

First, note that the structure is divided into "Standard fields" and "NT additional fields." The standard fields are those common to the Common Object File Format (COFF), which most UNIX executable files use. Though the standard fields retain the names defined in COFF, Windows NT actually uses some of them for different purposes that would be better described with other names.

- *Magic*. I was unable to track down what this field is used for. For the EXEVIEW.EXE sample application, the value is 0x010B or 267.

- *MajorLinkerVersion*, *MinorLinkerVersion*. Indicates version of the linker that linked this image. The preliminary Windows NT Software Development Kit (SDK), which shipped with build 438 of Windows NT, includes linker version 2.39 (2.27 hex).

- *SizeOfCode*. Size of executable code.

- *SizeOfInitializedData*. Size of initialized data.

- *SizeOfUninitializedData*. Size of uninitialized data.

- *AddressOfEntryPoint*. Of the standard fields, the *AddressOfEntryPoint* field is the most interesting for the PE file format. This field indicates the location of the entry point for the application and, perhaps more importantly to system hackers, the location of the end of the Import Address Table (IAT). The following function demonstrates how to retrieve the entry point of a Windows NT executable image from the optional header.

  PEFILE.C

```
LPVOID  WINAPI GetModuleEntryPoint (

    LPVOID    lpFile)

{

    PIMAGE_OPTIONAL_HEADER   poh;


    poh = (PIMAGE_OPTIONAL_HEADER)OPTHDROFFSET (lpFile);


    if (poh != NULL)

        return (LPVOID)poh->AddressOfEntryPoint;

    else

        return NULL;

}
```

- *BaseOfCode*. Relative offset of code (".text" section) in loaded image.

- *BaseOfData*. Relative offset of uninitialized data (".bss" section) in loaded image.

Windows NT Additional Fields

The additional fields added to the Windows NT PE file format provide loader support for much of the Windows NT-specific process behavior. Following is a summary of these fields.

- *ImageBase*. Preferred base address in the address space of a process to map the executable image to. The linker that comes with the Microsoft Win32 SDK for Windows NT defaults to 0x00400000, but you can override the default with the **-BASE: linker** switch.

- *SectionAlignment*. Each section is loaded into the address space of a process sequentially, beginning at *ImageBase*. *SectionAlignment* dictates the minimum amount of space a section can occupy when loaded—that is, sections are aligned on *SectionAlignment* boundaries.

  Section alignment can be no less than the page size (currently 4096 bytes on the *x*86 platform) and must be a multiple of the page size as dictated by the behavior of Windows NT's virtual memory manager. 4096 bytes is the *x*86 linker default, but this can be set using the **-ALIGN: linker** switch.

- *FileAlignment*. Minimum granularity of chunks of information within the image file prior to loading. For example, the linker zero-pads a section body (raw data for a section) up to the nearest *FileAlignment* boundary in the file. Version 2.39 of the linker mentioned earlier aligns image files on a 0x200-byte granularity. This value is constrained to be a power of 2 between 512 and 65,535.

- *MajorOperatingSystemVersion*. Indicates the major version of the Windows NT operating system, currently set to 1 for Windows NT version 1.0.

- *MinorOperatingSystemVersion*. Indicates the minor version of the Windows NT operating system, currently set to 0 for Windows NT version 1.0

- *MajorImageVersion*. Used to indicate the major version number of the application; in Microsoft Excel version 4.0, it would be 4.

- *MinorImageVersion*. Used to indicate the minor version number of the application; in Microsoft Excel version 4.0, it would be 0.

- *MajorSubsystemVersion*. Indicates the Windows NT Win32 subsystem major version number, currently set to 3 for Windows NT version 3.10.

- *MinorSubsystemVersion*. Indicates the Windows NT Win32 subsystem minor version number, currently set to 10 for Windows NT version 3.10.

- *Reserved1*. Unknown purpose, currently not used by the system and set to zero by the linker.

- *SizeOfImage*. Indicates the amount of address space to reserve in the address space for the loaded executable image. This number is influenced greatly by *SectionAlignment*. For example, consider a system having a fixed page size of 4096 bytes. If you have an executable with 11 sections, each less than 4096 bytes, aligned on a 65,536-byte boundary, the *SizeOfImage* field would be set to 11 * 65,536 = 720,896 (176 pages). The same file linked with 4096-byte alignment would result in 11 * 4096 = 45,056 (11 pages) for the *SizeOfImage* field. This is a simple example in which each section requires less than a page of memory. In reality, the linker determines the exact *SizeOfImage* by figuring each section individually. It first determines how many bytes the section

requires, then it rounds up to the nearest page boundary, and finally it rounds page count to the nearest *SectionAlignment* boundary. The total is then the sum of each section's individual requirement.

- *SizeOfHeaders*. This field indicates how much space in the file is used for representing all the file headers, including the MS-DOS header, PE file header, PE optional header, and PE section headers. The section bodies begin at this location in the file.

- *CheckSum*. A checksum value is used to validate the executable file at load time. The value is set and verified by the linker. The algorithm used for creating these checksum values is proprietary information and will not be published.

- *Subsystem*. Field used to identify the target subsystem for this executable. Each of the possible subsystem values are listed in the WINNT.H file immediately after the **IMAGE_OPTIONAL_HEADER** structure.

- *DllCharacteristics*. Flags used to indicate if a DLL image includes entry points for process and thread initialization and termination.

- *SizeOfStackReserve*, *SizeOfStackCommit*, *SizeOfHeapReserve*, *SizeOfHeapCommit*. These fields control the amount of address space to reserve and commit for the stack and default heap. Both the stack and heap have default values of 1 page committed and 16 pages reserved. These values are set with the linker switches **-STACKSIZE:** and **-HEAPSIZE:**.

- *LoaderFlags*. Tells the loader whether to break on load, debug on load, or the default, which is to let things run normally.

- *NumberOfRvaAndSizes*. This field identifies the length of the *DataDirectory* array that follows. It is important to note that this field is used to identify the size of the array, not the number of valid entries in the array.

- *DataDirectory*. The data directory indicates where to find other important components of executable information in the file. It is really nothing more than an array of **IMAGE_DATA_DIRECTORY** structures that are located at the end of the optional header structure. The current PE file format defines 16 possible data directories, 11 of which are now being used.

Data Directories

As defined in WINNT.H, the data directories are:

WINNT.H

```
// Directory Entries


// Export Directory
#define IMAGE_DIRECTORY_ENTRY_EXPORT         0
// Import Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT         1
// Resource Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE       2
// Exception Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION      3
// Security Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY       4
// Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_BASERELOC      5
// Debug Directory
#define IMAGE_DIRECTORY_ENTRY_DEBUG          6
// Description String
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT      7
// Machine Value (MIPS GP)
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR      8
// TLS Directory
#define IMAGE_DIRECTORY_ENTRY_TLS            9
// Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10
```

Each data directory is basically a structure defined as an **IMAGE_DATA_DIRECTORY**. And although data directory entries themselves are the same, each specific directory type is entirely unique. The definition of each defined data directory is described in "Predefined Sections" later in this article.

WINNT.H

```
typedef struct _IMAGE_DATA_DIRECTORY {

    ULONG   VirtualAddress;

    ULONG   Size;

} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Each data directory entry specifies the size and relative virtual address of the directory. To locate a particular directory, you determine the relative address from the data directory array in the optional header. Then use the virtual address to determine which section the directory is in. Once you determine which section contains the directory, the section header for that section is then used to find the exact file offset location of the data directory.

So to get a data directory, you first need to know about sections, which are described next. An example of how to locate data directories immediately follows this discussion.

PE File Sections

The PE file specification consists of the headers defined so far and a generic object called a *section*. Sections contain the content of the file, including code, data, resources, and other executable information. Each section has a header and a body (the raw data). Section headers are described below, but section bodies lack a rigid file structure. They can be organized in almost any way a linker wishes to organize them, as long as the header is filled with enough information to be able to decipher the data.

Section Headers

Section headers are located sequentially right after the optional header in the PE file format. Each section header is 40 bytes with no padding between them. Section headers are defined as in the following structure:

WINNT.H

```
#define IMAGE_SIZEOF_SHORT_NAME              8


typedef struct _IMAGE_SECTION_HEADER {
    UCHAR   Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            ULONG   PhysicalAddress;
            ULONG   VirtualSize;
    } Misc;
    ULONG   VirtualAddress;
    ULONG   SizeOfRawData;
    ULONG   PointerToRawData;
    ULONG   PointerToRelocations;
    ULONG   PointerToLinenumbers;
    USHORT  NumberOfRelocations;
    USHORT  NumberOfLinenumbers;
    ULONG   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

How do you go about getting section header information for a particular section? Since section headers are organized sequentially in no specific order, section headers must be located by name. The following function shows how to retrieve a section header from a PE image file given the name of the section:

PEFILE.C

```
BOOL    WINAPI GetSectionHdrByName (
    LPVOID                  lpFile,
    IMAGE_SECTION_HEADER    *sh,
    char                    *szSection)
{
    PIMAGE_SECTION_HEADER   psh;
```

```
    int                     nSections = NumOfSections (lpFile);
    int                     i;



    if ((psh = (PIMAGE_SECTION_HEADER)SECHDROFFSET (lpFile)) !=
         NULL)
        {
        /* find the section by name */
        for (i=0; i<nSections; i++)
            {
            if (!strcmp (psh->Name, szSection))
                {
                /* copy data to header */
                CopyMemory ((LPVOID)sh,
                            (LPVOID)psh,
                            sizeof (IMAGE_SECTION_HEADER));
                return TRUE;
                }
            else
                psh++;
            }
        }


    return FALSE;
}
```

The function simply locates the first section header via the **SECHDROFFSET** macro. Then the function loops through each section, comparing each section's name with the name of the section it's looking for, until it finds the right one. When the section is found, the function copies the data from the memory-mapped file to the structure passed in to the function. The fields of the **IMAGE_SECTION_HEADER** structure can then be accessed directly from the structure.

Section Header Fields

- *Name*. Each section header has a *name* field up to eight characters long, for which the first character must be a period.

- *PhysicalAddress* or *VirtualSize*. The second field is a union field that is not currently used.

- *VirtualAddress*. This field identifies the virtual address in the process's address space to which to load the section. The actual address is created by taking the value of this field and adding it to the *ImageBase* virtual address in the optional header structure. Keep in mind, though, that if this image file represents a DLL, there is no guarantee that the DLL will be loaded to the *ImageBase* location requested. So once the file is loaded into a process, the actual *ImageBase* value should be verified programmatically using **GetModuleHandle**.

- *SizeOfRawData*. This field indicates the *FileAlignment*-relative size of the section body. The actual size of the section body will be less than or equal to a multiple of *FileAlignment* in the file. Once the image is loaded into a process's address space, the size of the section body becomes less than or equal to a multiple of *SectionAlignment*.

- *PointerToRawData*. This is an offset to the location of the section body in the file.

- *PointerToRelocations*, *PointerToLinenumbers*, *NumberOfRelocations*, *NumberOfLinenumbers*. None of these fields are used in the PE file format.

- *Characteristics*. Defines the section characteristics. These values are found both in WINNT.H and in the Portable Executable Format specification located on this CD.

Value	Definition
0x00000020	Code section
0x00000040	Initialized data section
0x00000080	Uninitialized data section
0x04000000	Section cannot be cached
0x08000000	Section is not pageable
0x10000000	Section is shared
0x20000000	Executable section
0x40000000	Readable section
0x80000000	Writable section

Locating Data Directories

Data directories exist within the body of their corresponding data section. Typically, data directories are the first structure within the section body, but not out of necessity. For that reason, you need to retrieve

information from both the section header and optional header to locate a specific data directory.

To make this process easier, the following function was written to locate the data directory for any of the directories defined in WINNT.H:

PEFILE.C

```
LPVOID  WINAPI ImageDirectoryOffset (
        LPVOID    lpFile,
        DWORD     dwIMAGE_DIRECTORY)
{
    PIMAGE_OPTIONAL_HEADER   poh;
    PIMAGE_SECTION_HEADER    psh;
    int                      nSections = NumOfSections (lpFile);
    int                      i = 0;
    LPVOID                   VAImageDir;

    /* Must be 0 thru (NumberOfRvaAndSizes-1). */
    if (dwIMAGE_DIRECTORY >= poh->NumberOfRvaAndSizes)
        return NULL;

    /* Retrieve offsets to optional and section headers. */
    poh = (PIMAGE_OPTIONAL_HEADER)OPTHDROFFSET (lpFile);
    psh = (PIMAGE_SECTION_HEADER)SECHDROFFSET (lpFile);

    /* Locate image directory's relative virtual address. */
    VAImageDir = (LPVOID)poh->DataDirectory
                      [dwIMAGE_DIRECTORY].VirtualAddress;

    /* Locate section containing image directory. */
    while (i++<nSections)
        {
        if (psh->VirtualAddress <= (DWORD)VAImageDir &&
            psh->VirtualAddress +
```

```
                psh->SizeOfRawData > (DWORD)VAImageDir)
            break;
        psh++;
        }

    if (i > nSections)
        return NULL;

    /* Return image import directory offset. */
    return (LPVOID)(((int)lpFile +
                     (int)VAImageDir. psh->VirtualAddress) +
                    (int)psh->PointerToRawData);
}
```

The function begins by validating the requested data directory entry number. Then it retrieves pointers to the optional header and first section header. From the optional header, the function determines the data directory's virtual address, and it uses this value to determine within which section body the data directory is located. Once the appropriate section body has been identified, the specific location of the data directory is found by translating the relative virtual address of the data directory to a specific address into the file.

Predefined Sections

An application for Windows NT typically has the nine predefined sections named .text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata, and .debug. Some applications do not need all of these sections, while others may define still more sections to suit their specific needs. This behavior is similar to code and data segments in MS-DOS and Windows version 3.1. In fact, the way an application defines a unique section is by using the standard compiler directives for naming code and data segments or by using the name segment compiler option **-NT**—exactly the same way in which applications defined unique code and data segments in Windows version 3.1.

The following is a discussion of some of the more interesting sections common to typical Windows NT PE files.

Executable code section, .text

One difference between Windows version 3.1 and Windows NT is that the default behavior combines all code segments (as they are referred to in Windows version 3.1) into a single section called ".text" in Windows NT. Since Windows NT uses a page-based virtual memory management system, there is no advantage to separating code into distinct code segments. Consequently, having one large code section is easier to manage for both the operating system and the application developer.

The .text section also contains the entry point mentioned earlier. The IAT also lives in the .text section immediately before the module entry point. (The IAT's presence in the .text section makes sense because the table is really a series of jump instructions, for which the specific location to jump to is the fixed-up address.) When Windows NT executable images are loaded into a process's address space, the IAT is fixed up with the location of each imported function's physical address. In order to find the IAT in the .text section, the loader simply locates the module entry point and relies on the fact that the IAT occurs immediately before the entry point. And since each entry is the same size, it is easy to walk backward in the table to find its beginning.

Data sections, .bss, .rdata, .data

The .bss section represents uninitialized data for the application, including all variables declared as static within a function or source module.

The .rdata section represents read-only data, such as literal strings, constants, and debug directory information.

All other variables (except automatic variables, which appear on the stack) are stored in the .data section. Basically, these are application or module global variables.

Resources section, .rsrc

The .rsrc section contains resource information for a module. It begins with a resource directory structure like most other sections, but this section's data is further structured into a resource tree. The **IMAGE_RESOURCE_DIRECTORY**, shown below, forms the root and nodes of the tree.

WINNT.H

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {

    ULONG   Characteristics;

    ULONG   TimeDateStamp;

    USHORT  MajorVersion;

    USHORT  MinorVersion;

    USHORT  NumberOfNamedEntries;

    USHORT  NumberOfIdEntries;

} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

Looking at the directory structure, you won't find any pointer to the next nodes. Instead, there are two fields, *NumberOfNamedEntries* and *NumberOfIdEntries*, used to indicate how many entries are attached to the directory. By *attached*, I mean the directory entries follow immediately after the directory in the section data. The named entries appear first in ascending alphabetical order, followed by the ID entries in ascending numerical order.

A directory entry consists of two fields, as described in the following **IMAGE_RESOURCE_DIRECTORY_ENTRY** structure:

WINNT.H

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {

    ULONG   Name;

    ULONG   OffsetToData;

} IMAGE_RESOURCE_DIRECTORY_ENTRY, *PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

The two fields are used for different things depending on the level of the tree. The *Name* field is used to identify either a type of resource, a resource name, or a resource's language ID. The *OffsetToData* field is always used to point to a sibling in the tree, either a directory node or a leaf node.

Leaf nodes are the lowest node in the resource tree. They define the size and location of the actual resource data. Each leaf node is represented using the following **IMAGE_RESOURCE_DATA_ENTRY** structure:

WINNT.H

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {

    ULONG   OffsetToData;

    ULONG   Size;
```

```
    ULONG   CodePage;
    ULONG   Reserved;
} IMAGE_RESOURCE_DATA_ENTRY, *PIMAGE_RESOURCE_DATA_ENTRY;
```

The two fields *OffsetToData* and *Size* indicate the location and size of the actual resource data. Since this information is used primarily by functions once the application has been loaded, it makes more sense to make the *OffsetToData* field a relative virtual address. This is precisely the case. Interestingly enough, all other offsets, such as pointers from directory entries to other directories, are offsets relative to the location of the root node.

To make all of this a little clearer, consider Figure 2.

{ewc msdncd, EWGraphic, bsd23457 1 /a "SDKL.BMP"}

Figure 2. A simple resource tree structure

Figure 2 depicts a very simple resource tree containing only two resource objects, a menu, and a string table. Further, the menu and string table have only one item each. Yet, you can see how complicated the resource tree becomes—even with as few resources as this.

At the root of the tree, the first directory has one entry for each type of resource the file contains, no matter how many of each type there are. In Figure 2, there are two entries identified by the root, one for the menu and one for the string table. If there had been one or more dialog resources included in the file, the root node would have had one more entry and, consequently, another branch for the dialog resources.

The basic resource types are identified in the file WINUSER.H and are listed below:

WINUSER.H

```
/*
 * Predefined Resource Types
 */
#define RT_CURSOR           MAKEINTRESOURCE(1)
#define RT_BITMAP           MAKEINTRESOURCE(2)
#define RT_ICON             MAKEINTRESOURCE(3)
#define RT_MENU             MAKEINTRESOURCE(4)
#define RT_DIALOG           MAKEINTRESOURCE(5)
#define RT_STRING           MAKEINTRESOURCE(6)
#define RT_FONTDIR          MAKEINTRESOURCE(7)
#define RT_FONT             MAKEINTRESOURCE(8)
#define RT_ACCELERATOR      MAKEINTRESOURCE(9)
#define RT_RCDATA           MAKEINTRESOURCE(10)
#define RT_MESSAGETABLE     MAKEINTRESOURCE(11)
```

At the top level of the tree, the MAKEINTRESOURCE values listed above are placed in the *Name* field of

each type entry, identifying the different resources by type.

Each of the entries in the root directory points to a sibling node in the second level of the tree. These nodes are directories, too, each having their own entries. At this level, the directories are used to identify the name of each resource within a given type. If you had multiple menus defined in your application, there would be an entry for each one here at the second level of the tree.

As you are probably already aware, resources can be identified by name or by integer. They are distinguished in this level of the tree via the *Name* field in the directory structure. If the most significant bit of the *Name* field is set, the other 31 bits are used as an offset to an **IMAGE_RESOURCE_DIR_STRING_U** structure.

WINNT.H

```
typedef struct _IMAGE_RESOURCE_DIR_STRING_U {

    USHORT  Length;

    WCHAR   NameString[ 1 ];

} IMAGE_RESOURCE_DIR_STRING_U, *PIMAGE_RESOURCE_DIR_STRING_U;
```

This structure is simply a 2-byte *Length* field followed by *Length* UNICODE characters.

On the other hand, if the most significant bit of the *Name* field is clear, the lower 31 bits are used to represent the integer ID of the resource. Figure 2 shows the menu resource as a named resource and the string table as an ID resource.

If there were two menu resources, one identified by name and one by resource, they would both have entries immediately after the menu resource directory. The named resource entry would appear first, followed by the integer-identified resource. The directory fields *NumberOfNamedEntries* and *NumberOfIdEntries* would each contain the value 1, indicating the presence of one entry.

Below level two, the resource tree does not branch out any further. Level one branches into directories representing each type of resource, and level two branches into directories representing each resource by identifier. Level three maps a one-to-one correspondence between the individually identified resources and their respective language IDs. To indicate the language ID of a resource, the *Name* field of the directory entry structure is used to indicate both the primary language and sublanguage ID for the resource. The Win32 SDK for Windows NT lists the default value resources. For the value 0x0409, 0x09 represents the primary language as LANG_ENGLISH, and 0x04 is defined as SUBLANG_ENGLISH_CAN for the sublanguage. The entire set of language IDs is defined in the file WINNT.H, included as part of the Win32 SDK for Windows NT.

Since the language ID node is the last directory node in the tree, the *OffsetToData* field in the entry structure is an offset to a leaf node—the **IMAGE_RESOURCE_DATA_ENTRY** structure mentioned earlier.

Referring back to Figure 2, you can see one data entry node for each language directory entry. This node simply indicates the size of the resource data and the relative virtual address where the resource data is located.

One advantage to having so much structure to the resource data section, .rsrc, is that you can glean a great deal of information from the section without accessing the resources themselves. For example, you can find out how many there are of each type of resource, what resources—if any—use a particular language ID, whether a particular resource exists or not, and the size of individual types of resources. To demonstrate how to make use of this information, the following function shows how to determine the different types of resources a file includes:

PEFILE.C

```
int     WINAPI GetListOfResourceTypes (
    LPVOID    lpFile,
    HANDLE    hHeap,
    char      **pszResTypes)
{
    PIMAGE_RESOURCE_DIRECTORY          prdRoot;
    PIMAGE_RESOURCE_DIRECTORY_ENTRY    prde;
    char                               *pMem;
    int                                nCnt, i;


    /* Get root directory of resource tree. */
    if ((prdRoot = PIMAGE_RESOURCE_DIRECTORY)ImageDirectoryOffset
           (lpFile, IMAGE_DIRECTORY_ENTRY_RESOURCE)) == NULL)
        return 0;

    /* Allocate enough space from heap to cover all types. */
    nCnt = prdRoot->NumberOfIdEntries * (MAXRESOURCENAME + 1);
    *pszResTypes = (char *)HeapAlloc (hHeap,
                                      HEAP_ZERO_MEMORY,
                                      nCnt);
    if ((pMem = *pszResTypes) == NULL)
        return 0;

    /* Set pointer to first resource type entry. */
    prde = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD)prdRoot +
              sizeof (IMAGE_RESOURCE_DIRECTORY));

    /* Loop through all resource directory entry types. */
    for (i=0; i<prdRoot->NumberOfIdEntries; i++)
        {
```

```
            if (LoadString (hDll, prde->Name, pMem, MAXRESOURCENAME))
                pMem += strlen (pMem) + 1;

            prde++;
            }

        return nCnt;
    }
```

This function returns a list of resource type names in the string identified by *pszResTypes*. Notice that, at the heart of this function, **LoadString** is called using the *Name* field of each resource type directory entry as the string ID. If you look in the PEFILE.RC, you'll see that I defined a series of resource type strings whose IDs are defined the same as the type specifiers in the directory entries. There is also a function in PEFILE.DLL that returns the total number of resource objects in the .rsrc section. It would be rather easy to expand on these functions or write new functions that extracted other information from this section.

Export data section, .edata

The .edata section contains export data for an application or DLL. When present, this section contains an export directory for getting to the export information.

WINNT.H

```
typedef struct _IMAGE_EXPORT_DIRECTORY {

    ULONG   Characteristics;

    ULONG   TimeDateStamp;

    USHORT  MajorVersion;

    USHORT  MinorVersion;

    ULONG   Name;

    ULONG   Base;

    ULONG   NumberOfFunctions;

    ULONG   NumberOfNames;

    PULONG  *AddressOfFunctions;

    PULONG  *AddressOfNames;

    PUSHORT *AddressOfNameOrdinals;

} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

The *Name* field in the export directory identifies the name of the executable module. *NumberOfFunctions* and *NumberOfNames* fields indicate how many functions and function names are being exported from the module.

The *AddressOfFunctions* field is an offset to a list of exported function entry points. The *AddressOfNames* field is the address of an offset to the beginning of a null-separated list of exported function names. *AddressOfNameOrdinals* is an offset to a list of ordinal values (each 2 bytes long) for the same exported functions.

The three *AddressOf...* fields are relative virtual addresses into the address space of a process once the module has been loaded. Once the module is loaded, the relative virtual address should be added to the module base address to get the exact location in the address space of the process. Before the file is loaded, however, the address can be determined by subtracting the section header virtual address (*VirtualAddress*) from the given field address, adding the section body offset (*PointerToRawData*) to the result, and then using this value as an offset into the image file. The following example illustrates this technique:

PEFILE.C

```
int  WINAPI GetExportFunctionNames (

    LPVOID    lpFile,

    HANDLE    hHeap,

    char      **pszFunctions)
```

```
{
    IMAGE_SECTION_HEADER       sh;
    PIMAGE_EXPORT_DIRECTORY    ped;
    char                       *pNames, *pCnt;
    int                        i, nCnt;

    /* Get section header and pointer to data directory
       for .edata section. */
    if ((ped = (PIMAGE_EXPORT_DIRECTORY)ImageDirectoryOffset
            (lpFile, IMAGE_DIRECTORY_ENTRY_EXPORT)) == NULL)
        return 0;
    GetSectionHdrByName (lpFile, &sh, ".edata");

    /* Determine the offset of the export function names. */
    pNames = (char *)(*(int *)((int)ped->AddressOfNames -
                               (int)sh.VirtualAddress   +
                               (int)sh.PointerToRawData +
                               (int)lpFile)    -
                      (int)sh.VirtualAddress   +
                      (int)sh.PointerToRawData +
                      (int)lpFile);

    /* Figure out how much memory to allocate for all strings. */
    pCnt = pNames;
    for (i=0; i<(int)ped->NumberOfNames; i++)
        while (*pCnt++);
    nCnt = (int)(pCnt. pNames);

    /* Allocate memory off heap for function names. */
    *pszFunctions = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nCnt);
```

```
    /* Copy all strings to buffer. */

    CopyMemory ((LPVOID)*pszFunctions, (LPVOID)pNames, nCnt);


    return nCnt;

}
```

Notice that in this function the variable *pNames* is assigned by determining first the address of the offset and then the actual offset location. Both the address of the offset and the offset itself are relative virtual addresses and must be translated before being used, as the function demonstrates. You could write a similar function to determine the ordinal values or entry points of the functions, but why bother when I already did this for you? The **GetNumberOfExportedFunctions**, **GetExportFunctionEntryPoints**, and **GetExportFunctionOrdinals** functions also exist in the PEFILE.DLL.

Import data section, .idata

The .idata section is import data, including the import directory and import address name table. Although an IMAGE_DIRECTORY_ENTRY_IMPORT directory is defined, no corresponding import directory structure is included in the file WINNT.H. Instead, there are several other structures called IMAGE_IMPORT_BY_NAME, IMAGE_THUNK_DATA, and IMAGE_IMPORT_DESCRIPTOR. Personally, I couldn't make heads or tails of how these structures are supposed to correlate to the .idata section, so I spent several hours deciphering the .idata section body and came up with a much simpler structure. I named this structure **IMAGE_IMPORT_MODULE_DIRECTORY**.

PEFILE.H

```
typedef struct tagImportDirectory

    {

    DWORD    dwRVAFunctionNameList;

    DWORD    dwUseless1;

    DWORD    dwUseless2;

    DWORD    dwRVAModuleName;

    DWORD    dwRVAFunctionAddressList;

    }IMAGE_IMPORT_MODULE_DIRECTORY,

     * PIMAGE_IMPORT_MODULE_DIRECTORY;
```

Unlike the data directories of other sections, this one repeats one after another for each imported module in the file. Think of it as an entry in a list of module data directories, rather than a data directory to the entire section of data. Each entry is a directory to the import information for a specific module.

One of the fields in the **IMAGE_IMPORT_MODULE_DIRECTORY** structure is *dwRVAModuleName*, a relative virtual address pointing to the name of the module. There are also two *dwUseless* parameters in the structure that serve as padding to keep the structure aligned properly within the section. The PE file format specification mentions something about import flags, a time/date stamp, and major/minor versions, but these two fields remained empty throughout my experimentation, so I still consider them useless.

Based on the definition of this structure, you can retrieve the names of modules and all functions in each module that are imported by an executable file. The following function demonstrates how to retrieve all the module names imported by a particular PE file:

PEFILE.C

```
int  WINAPI GetImportModuleNames (

    LPVOID    lpFile,

    HANDLE    hHeap,

    char      **pszModules)

{

    PIMAGE_IMPORT_MODULE_DIRECTORY  pid;

    IMAGE_SECTION_HEADER            idsh;
```

```
BYTE                            *pData;
int                             nCnt = 0, nSize = 0, i;
char                            *pModule[1024];
char                            *psz;

pid = (PIMAGE_IMPORT_MODULE_DIRECTORY)ImageDirectoryOffset
         (lpFile, IMAGE_DIRECTORY_ENTRY_IMPORT);
pData = (BYTE *)pid;

/* Locate section header for ".idata" section. */
if (!GetSectionHdrByName (lpFile, &idsh, ".idata"))
    return 0;

/* Extract all import modules. */
while (pid->dwRVAModuleName)
    {
    /* Allocate buffer for absolute string offsets. */
    pModule[nCnt] = (char *)(pData +
           (pid->dwRVAModuleName-idsh.VirtualAddress));
    nSize += strlen (pModule[nCnt]) + 1;

    /* Increment to the next import directory entry. */
    pid++;
    nCnt++;
    }

/* Copy all strings to one chunk of heap memory. */
*pszModules = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nSize);
psz = *pszModules;
for (i=0; i<nCnt; i++)
    {
```

```
            strcpy (psz, pModule[i]);
            psz += strlen (psz) + 1;
            }

        return nCnt;
    }
```

The function is pretty straightforward. However, one thing is worth pointing out—notice the while loop. This loop is terminated when *pid->dwRVAModuleName* is 0. Implied here is that at the end of the list of **IMAGE_IMPORT_MODULE_DIRECTORY** structures is a null structure that has a value of 0 for at least the *dwRVAModuleName* field. This is the behavior I observed in my experimentation with the file and later confirmed in the PE file format specification.

The first field in the structure, *dwRVAFunctionNameList*, is a relative virtual address to a list of relative virtual addresses that each point to the function names within the file. As shown in the following data, the module and function names of all imported modules are listed in the .idata section data:

```
E6A7 0000 F6A7 0000  08A8 0000 1AA8 0000  ................
28A8 0000 3CA8 0000  4CA8 0000 0000 0000  (...<...L.......
0000 4765 744F 7065  6E46 696C 654E 616D  ..GetOpenFileNam
6541 0000 636F 6D64  6C67 3332 2E64 6C6C  eA..comdlg32.dll
0000 2500 4372 6561  7465 466F 6E74 496E  ..%.CreateFontIn
6469 7265 6374 4100  4744 4933 322E 646C  directA.GDI32.dl
6C00 A000 4765 7444  6576 6963 6543 6170  l...GetDeviceCap
7300 C600 4765 7453  746F 636B 4F62 6A65  s...GetStockObje
6374 0000 D500 4765  7454 6578 744D 6574  ct....GetTextMet
7269 6373 4100 1001  5365 6C65 6374 4F62  ricsA...SelectOb
6A65 6374 0000 1601  5365 7442 6B43 6F6C  ject....SetBkCol
6F72 0000 3501 5365  7454 6578 7443 6F6C  or..5.SetTextCol
6F72 0000 4501 5465  7874 4F75 7441 0000  or..E.TextOutA..
```

The above data is a portion taken from the .idata section of the EXEVIEW.EXE sample application. This particular section represents the beginning of the list of import module and function names. If you begin examining the right section part of the data, you should recognize the names of familiar Win32 API functions and the module names they are found in. Reading from the top down, you get **GetOpenFileNameA**, followed by the module name COMDLG32.DLL. Shortly after that, you get **CreateFontIndirectA**, followed by the module GDI32.DLL and then the functions **GetDeviceCaps**, **GetStockObject**, **GetTextMetrics**, and so forth.

This pattern repeats throughout the .idata section. The first module name is COMDLG32.DLL and the second is GDI32.DLL. Notice that only one function is imported from the first module, while many functions are imported from the second module. In both cases, the function names and the module name

to which they belong are ordered such that a function name appears first, followed by the module name and then by the rest of the function names, if any.

The following function demonstrates how to retrieve the function names for a specific module:

PEFILE.C

```
int  WINAPI GetImportFunctionNamesByModule (
    LPVOID    lpFile,
    HANDLE    hHeap,
    char      *pszModule,
    char      **pszFunctions)
{
    PIMAGE_IMPORT_MODULE_DIRECTORY  pid;
    IMAGE_SECTION_HEADER     idsh;
    DWORD                    dwBase;
    int                      nCnt = 0, nSize = 0;
    DWORD                    dwFunction;
    char                     *psz;


    /* Locate section header for ".idata" section. */
    if (!GetSectionHdrByName (lpFile, &idsh, ".idata"))
        return 0;

    pid = (PIMAGE_IMPORT_MODULE_DIRECTORY)ImageDirectoryOffset
             (lpFile, IMAGE_DIRECTORY_ENTRY_IMPORT);

    dwBase = ((DWORD)pid. idsh.VirtualAddress);

    /* Find module's pid. */
    while (pid->dwRVAModuleName &&
           strcmp (pszModule,
                  (char *)(pid->dwRVAModuleName+dwBase)))
```

```
    pid++;

/* Exit if the module is not found. */
if (!pid->dwRVAModuleName)
    return 0;

/* Count number of function names and length of strings. */
dwFunction = pid->dwRVAFunctionNameList;
while (dwFunction                       &&
       *(DWORD *)(dwFunction + dwBase) &&
       *(char *)((*(DWORD *)(dwFunction + dwBase)) +
        dwBase+2))
    {
    nSize += strlen ((char *)((*(DWORD *)(dwFunction +
         dwBase)) + dwBase+2)) + 1;
    dwFunction += 4;
    nCnt++;
    }

/* Allocate memory off heap for function names. */
*pszFunctions = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nSize);
psz = *pszFunctions;

/* Copy function names to memory pointer. */
dwFunction = pid->dwRVAFunctionNameList;
while (dwFunction                       &&
       *(DWORD *)(dwFunction + dwBase) &&
       *((char *)((*(DWORD *)(dwFunction + dwBase)) +
        dwBase+2)))
    {
    strcpy (psz, (char *)((*(DWORD *)(dwFunction + dwBase)) +
```

```
                dwBase+2));
            psz += strlen((char *)((*(DWORD *)(dwFunction + dwBase))+
                dwBase+2)) + 1;
            dwFunction += 4;
            }

        return nCnt;
    }
```

Like the **GetImportModuleNames** function, this function relies on the end of each list of information to have a zeroed entry. In this case, the list of function names ends with one that is zero.

The final field, *dwRVAFunctionAddressList*, is a relative virtual address to a list of virtual addresses that will be placed in the section data by the loader when the file is loaded. Before the file is loaded, however, these virtual addresses are replaced by relative virtual addresses that correspond exactly to the list of function names. So before the file is loaded, there are two identical lists of relative virtual addresses pointing to imported function names.

Debug information section, .debug

Debug information is initially placed in the .debug section. The PE file format also supports separate debug files (normally identified with a .DBG extension) as a means of collecting debug information in a central location. The debug section contains the debug information, but the debug directories live in the .rdata section mentioned earlier. Each of those directories references debug information in the .debug section. The debug directory structure is defined as an **IMAGE_DEBUG_DIRECTORY**, as follows:

WINNT.H

```
typedef struct _IMAGE_DEBUG_DIRECTORY {

    ULONG   Characteristics;

    ULONG   TimeDateStamp;

    USHORT  MajorVersion;

    USHORT  MinorVersion;

    ULONG   Type;

    ULONG   SizeOfData;

    ULONG   AddressOfRawData;

    ULONG   PointerToRawData;

} IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;
```

The section is divided into separate portions of data representing different types of debug information. For each one there is a debug directory described above. The different types of debug information are listed below:

WINNT.H

```
#define IMAGE_DEBUG_TYPE_UNKNOWN          0

#define IMAGE_DEBUG_TYPE_COFF             1

#define IMAGE_DEBUG_TYPE_CODEVIEW         2

#define IMAGE_DEBUG_TYPE_FPO              3

#define IMAGE_DEBUG_TYPE_MISC             4
```

The *Type* field in each directory indicates which type of debug information the directory represents. As you can see in the list above, the PE file format supports many different types of debug information, as well as some other informational fields. Of those, the **IMAGE_DEBUG_TYPE_MISC** information is unique. This information was added to represent miscellaneous information about the executable image that could not be added to any of the more structured data sections in the PE file format. This is the only location in the image file where the image name is sure to appear. If an image exports information, the export data section will also include the image name.

Each type of debug information has its own header structure that defines its data. Each of these is listed in the file WINNT.H. One nice thing about the **IMAGE_DEBUG_DIRECTORY** structure is that it includes two fields that identify the debug information. The first of these, *AddressOfRawData*, is the relative virtual address of the data once the file is loaded. The other, *PointerToRawData*, is an actual offset within the PE file, where the data is located. This makes it easy to locate specific debug information.

As a last example, consider the following function, which extracts the image name from the **IMAGE_DEBUG_MISC** structure:

PEFILE.C

```
int    WINAPI RetrieveModuleName (
    LPVOID    lpFile,
    HANDLE    hHeap,
    char      **pszModule)
{

    PIMAGE_DEBUG_DIRECTORY    pdd;
    PIMAGE_DEBUG_MISC         pdm = NULL;
    int                       nCnt;

    if (!(pdd = (PIMAGE_DEBUG_DIRECTORY)ImageDirectoryOffset
            (lpFile, IMAGE_DIRECTORY_ENTRY_DEBUG)))
        return 0;

    while (pdd->SizeOfData)
        {
        if (pdd->Type == IMAGE_DEBUG_TYPE_MISC)
            {
            pdm = (PIMAGE_DEBUG_MISC)
                ((DWORD)pdd->PointerToRawData + (DWORD)lpFile);

            nCnt = lstrlen (pdm->Data)*(pdm->Unicode?2:1);
            *pszModule = (char *)HeapAlloc (hHeap,
                                            HEAP_ZERO_MEMORY,
                                            nCnt+1;
            CopyMemory (*pszModule, pdm->Data, nCnt);

            break;
```

```
            }


        pdd ++;

        }


    if (pdm != NULL)

        return nCnt;

    else

        return 0;

}
```

As you can see, the structure of the debug directory makes it relatively easy to locate a specific type of debug information. Once the **IMAGE_DEBUG_MISC** structure is located, extracting the image name is as simple as invoking the **CopyMemory** function.

As mentioned above, debug information can be stripped into separate .DBG files. The Windows NT SDK includes a utility called REBASE.EXE that serves this purpose. For example, in the following statement an executable image named TEST.EXE is being stripped of debug information:

```
rebase -b 40000 -x c:\samples\testdir test.exe
```

The debug information is placed in a new file called TEST.DBG and located in the path specified, in this case c:\samples\testdir. The file begins with a single **IMAGE_SEPARATE_DEBUG_HEADER** structure, followed by a copy of the section headers that exist in the stripped executable image. Then the .debug section data follows the section headers. So, right after the section headers are the series of **IMAGE_DEBUG_DIRECTORY** structures and their associated data. The debug information itself retains the same structure as described above for normal image file debug information.

Summary of the PE File Format

The PE file format for Windows NT introduces a completely new structure to developers familiar with the Windows and MS-DOS environments. Yet developers familiar with the UNIX environment will find that the PE file format is similar to, if not based on, the COFF specification.

The entire format consists of an MS-DOS MZ header, followed by a real-mode stub program, the PE file signature, the PE file header, the PE optional header, all of the section headers, and finally, all of the section bodies.

The optional header ends with an array of data directory entries that are relative virtual addresses to data directories contained within section bodies. Each data directory indicates how a specific section body's data is structured.

The PE file format has eleven predefined sections, as is common to applications for Windows NT, but each application can define its own unique sections for code and data.

The .debug predefined section also has the capability of being stripped from the file into a separate debug file. If so, a special debug header is used to parse the debug file, and a flag is specified in the PE file header to indicate that the debug data has been stripped.

PEFILE.DLL Function Descriptions

PEFILE.DLL consists mainly of functions that either retrieve an offset into a given PE file or copy a portion of the file data to a specific structure. Each function has a single requirement—the first parameter is a pointer to the beginning of the PE file. That is, the file must first be memory-mapped into the address space of your process, and the base location of the file mapping is the value *lpFile* that you pass as the first parameter to every function.

The function names are meant to be self-explanatory, and each function is listed with a brief comment describing its purpose. If, after reading through the list of functions, you cannot determine what a function is for, refer to the EXEVIEW.EXE sample application to find an example of how the function is used. The following list of function prototypes can also be found in PEFILE.H:

PEFILE.H

```
/* Retrieve a pointer offset to the MS-DOS MZ header. */
BOOL WINAPI GetDosHeader (LPVOID, PIMAGE_DOS_HEADER);


/* Determine the type of an .EXE file. */
DWORD WINAPI ImageFileType (LPVOID);


/* Retrieve a pointer offset to the PE file header. */
BOOL WINAPI GetPEFileHeader (LPVOID, PIMAGE_FILE_HEADER);


/* Retrieve a pointer offset to the PE optional header .*/
BOOL WINAPI GetPEOptionalHeader (LPVOID,
                                 PIMAGE_OPTIONAL_HEADER);


/* Return the address of the module entry point. */
LPVOID WINAPI GetModuleEntryPoint (LPVOID);


/* Return a count of the number of sections in the file. */
int  WINAPI NumOfSections (LPVOID);


/* Return the desired base address of the executable when
   it is loaded into a process's address space. */
LPVOID WINAPI GetImageBase (LPVOID);
```

```
/* Determine the location within the file of a specific
   image data directory.  */
LPVOID WINAPI ImageDirectoryOffset (LPVOID, DWORD);

/* Function retrieve names of all the sections in the file. */
int WINAPI GetSectionNames (LPVOID, HANDLE, char **);

/* Copy the section header information for a specific section. */
BOOL WINAPI GetSectionHdrByName (LPVOID,
                                 PIMAGE_SECTION_HEADER, char *);

/* Get null-separated list of import module names. */
int WINAPI GetImportModuleNames (LPVOID, HANDLE, char  **);

/* Get null-separated list of import functions for a module. */
int WINAPI GetImportFunctionNamesByModule (LPVOID, HANDLE,
                                           char *, char  **);

/* Get null-separated list of exported function names. */
int WINAPI GetExportFunctionNames (LPVOID, HANDLE, char **);

/* Get number of exported functions. */
int WINAPI GetNumberOfExportedFunctions (LPVOID);

/* Get list of exported function virtual address entry points. */
LPVOID WINAPI GetExportFunctionEntryPoints (LPVOID);

/* Get list of exported function ordinal values. */
LPVOID WINAPI GetExportFunctionOrdinals (LPVOID);
```

```
/* Determine total number of resource objects. */
int WINAPI GetNumberOfResources (LPVOID);

/* Return list of all resource object types used in file. */
int WINAPI GetListOfResourceTypes (LPVOID, HANDLE, char **);

/* Determine if debug information has been removed from file. */
BOOL WINAPI IsDebugInfoStripped (LPVOID);

/* Get name of image file. */
int WINAPI RetrieveModuleName (LPVOID, HANDLE, char **);

/* Function determines if the file is a valid debug file. */
BOOL WINAPI IsDebugFile (LPVOID);

/* Function returns debug header from debug file. */
BOOL WINAPI GetSeparateDebugHeader(LPVOID,
                                   PIMAGE_SEPARATE_DEBUG_HEADER);
```

In addition to the functions listed above, the macros mentioned earlier in this article are also defined in the PEFILE.H file. The complete list is as follows:

```
/* Offset to PE file signature                                */
#define NTSIGNATURE(a) ((LPVOID)((BYTE *)a                +  \
                        ((PIMAGE_DOS_HEADER)a)->e_lfanew))

/* MS-OS header identifies the NT PEFile signature dword;
   the PEFILE header exists just after that dword.           */
#define PEFHDROFFSET(a) ((LPVOID)((BYTE *)a               +  \
                         ((PIMAGE_DOS_HEADER)a)->e_lfanew +  \
                             SIZE_OF_NT_SIGNATURE))

/* PE optional header is immediately after PEFile header.    */
```

```
#define OPTHDROFFSET(a) ((LPVOID)((BYTE *)a              +  \
                         ((PIMAGE_DOS_HEADER)a)->e_lfanew +  \
                           SIZE_OF_NT_SIGNATURE           +  \
                           sizeof (IMAGE_FILE_HEADER)))


/* Section headers are immediately after PE optional header. */
#define SECHDROFFSET(a) ((LPVOID)((BYTE *)a              +  \
                         ((PIMAGE_DOS_HEADER)a)->e_lfanew +  \
                           SIZE_OF_NT_SIGNATURE           +  \
                           sizeof (IMAGE_FILE_HEADER)     +  \
                           sizeof (IMAGE_OPTIONAL_HEADER)))
```

To use PEFILE.DLL, simply include the header file PEFILE.H and link the DLL to your application. All of the functions are mutually exclusive functions, but some were written as much to support others as for the information they provide. For example, the function **GetSectionNames** is useful for getting the exact names of all sections. Yet to be able to retrieve the section header for a unique section name (one defined by the application developer during compile), you would first have to get the list of names and then call the function **GetSectionHeaderByName** with the exact name of the section. Enjoy!

Writing Scalable Applications for Windows NT

John Vert
Windows NT Base Group

Revision 1.0: June 6, 1995

1. Introduction

One of the major goals of Microsoft® Windows NT® is to provide a robust, scalable symmetric multiprocessing (SMP) operating system. By harnessing the power of multiple processors, SMP can deliver a large boost to the performance and capacity of both workstation and server applications. Scalable SMP client-server systems can be very effective platforms for downsizing traditional mainframe applications. SMP also makes it easy to increase the computing capacity of a heavily loaded server by adding more processors. Windows NT provides a great platform to base server applications on, but the operating system is not solely responsible for performance and scalability. Applications hosted on Windows NT must also be designed with these goals in mind. A perfectly efficient system devotes all of its resources to any given problem. But without a carefully designed application, unnecessary system code will be executed, leaving fewer resources available for the application to devote to the problem.

Windows NT provides many advanced features to make the development of efficient, scalable applications easier. Other SMP operating systems have some of these features, but a few are unique to Windows NT. Understanding and using these features is the key to realizing the full potential of Windows NT SMP in your application. This article will cover the use of these features, and describe some of the common pitfalls encountered in SMP programming. It assumes you have a good working knowledge of Win32®, particularly overlapped input/output (I/O) and network programming.

2. Threads

Windows NT's basic unit of execution is the *thread*. Each process contains one or more threads. Threads can run on any processor in a multiprocessor system, so splitting a single-threaded program into multiple concurrent threads is a quick way to take advantage of SMP systems. A similar approach splits a single-threaded server into multiple server processes. Traditional UNIX® SMP operating systems that do not have native support for threads often use this approach. Since processes require more system overhead than threads, a single multithreaded program is a more efficient solution on Windows NT.

In order to effectively split a single-threaded program into multiple threads, you need to understand how threads work. Normally, a thread can be in one of three states at a given time.

- **Waiting**—The thread cannot run until a specified event occurs.
- **Ready**—The thread is ready to run, but no processor is currently available.
- **Running**—The thread is currently running on a processor.

Any thread in either the ready or running state is runnable and may profitably use any available control processing unit (CPU) cycles. The number of runnable threads is limited only by system resources, but the number of currently running threads is limited by the number of processors in the system.

2.1 The Windows NT Scheduler

The Windows NT kernel is responsible for allocating the available CPUs among the system's runnable threads in the most efficient manner. To do this, Windows NT uses a priority-based round-robin algorithm. The Windows NT kernel supports 31 different priorities, and a queue for each of the 31 priorities contains all the ready threads at that priority. When a CPU becomes available, the kernel finds the highest priority queue with ready threads on it, removes the thread at the head of the queue, and runs it. This process is called a *context switch.*

The most common reason for a context switch is when a running thread needs to wait. This happens for a number of different reasons. If a thread touches a page that is not in its working set, the thread must wait for memory management to resolve the page fault before it can continue. Many system calls, such as **WaitForSingleObject** or **ReadFile**, explicitly block the running thread until the specified event occurs.

When a running thread needs to wait, the kernel picks the highest-priority ready thread and switches it from the ready state to the running state. This ensures that the highest priority runnable threads are always running. To prevent CPU-bound threads from monopolizing the processor, the kernel imposes a time limit (called the *thread quantum*) on each thread. When a thread has been running for one quantum, the kernel preempts it and moves it to the end of the ready queue for its priority. The actual length of a thread's quantum currently varies from 15 milliseconds to 30 milliseconds across different Windows NT platforms, but this may change in future versions.

Another reason for a context switch is when an event changes a higher-priority thread's state from waiting to ready. In this case, the higher-priority thread will immediately preempt a lower-priority thread running on the processor.

On a uniprocessor computer, only one thread can be in the running state, because there is only one processor. A multiprocessor computer allows for one running thread per processor. It is important to understand that a multiprocessor computer will *not* make a single thread complete its activity any faster. The entire performance gain is a result of multiple threads running simultaneously. Even if your application uses multiple threads, the threads must be able to work independently of each other to scale effectively. If your application is too serialized (meaning that threads have interdependencies that force them to wait for each other) there will not be enough runnable threads to distribute across the processors, and some processors may be idle. Adding more processors will only increase the total CPU time spent idle— it will not make your application run much faster.

Another factor to consider is whether your application is *compute-bound* (limited by the speed of the CPU) or *I/O-bound* (limited by the speed of some I/O device, typically disk drivers or network bandwidth). If your application is I/O bound and cannot saturate the CPU of a uniprocessor computer, adding more processors is unlikely to make it run much faster. If your application spends most of its time waiting for the disk, additional CPUs will just tighten the I/O bottleneck.

2.1 How Many Threads Do I Need?

There are two basic models for implementing a client-server application. The easiest to use is a single thread that services all client requests in turn. However, this model suffers from the "too few threads" syndrome, and will not work any faster on an SMP computer. It's not even a very good model for a uniprocessor computer, because if the single thread ever blocks, no application work can be done. Blocking on I/O, for example, is almost inevitable in any application. This model can be stretched to one thread per processor, but the problem of evenly dividing client requests among the threads is difficult to solve efficiently. If one of the threads needs to wait for I/O, there is still no way to prevent its CPU from idling until the wait completes.

The model at the other end of the spectrum creates one thread for each client. This readily solves the problem of providing enough threads to utilize all the CPUs. Because each client has its own dedicated thread, there is no need to manually balance the entire client load over a few threads. But this is an expensive solution that degrades with large numbers of clients. As the number of ready threads becomes much greater than the number of processors, overall performance decreases. Each thread spends more time waiting on the ready queue for its turn to run, and the kernel must spend more time context switching the threads in and out of the running state.

Threads are not free, so a design that uses hundreds of ready threads can consume quite a lot of system resources in the form of memory and increased scheduling overhead. Windows NT can swap out the memory resources used by a waiting thread, but before a thread can become runnable, its resources must be brought back into memory.

3. I/O Completion Ports

An ideal model would strike a balance between the two extremes. There should always be enough runnable threads to fully utilize the available CPUs, but there should never be so many threads that the overhead becomes too large. In fact, the ideal number of runnable threads is not related to the number of clients at all, but to the number of CPUs in the server. Unfortunately, multiplexing a large number of clients across a smaller number of runnable threads is difficult for an application to do. The application cannot always know when a given thread is going to block, and without this knowledge it cannot activate another thread to take its place. To solve this problem and make it easy for programmers to write efficient, scalable applications, Windows NT version 3.5 provides a new mechanism called the *I/O completion port*.

An I/O completion port is designed for use with overlapped I/O. **CreateIoCompletionPort** associates the port with multiple file handles. When asynchronous I/O initiated on any of these file handles completes, an I/O completion packet is queued to the port. This combines the synchronization point for multiple file handles into a single object. If each file handle represents a connection to a client (usually through a named pipe or socket), then a handful of threads can manage I/O for any number of clients by waiting on the I/O completion port. Rather than directly waiting for overlapped I/O to complete, these threads use **GetQueuedCompletionStatus** to wait on the I/O completion port. Any thread that waits on a completion port becomes associated with that port. The Windows NT kernel keeps track of the threads associated with an I/O completion port.

Of course, **WaitForMultipleObjects** can produce similar behavior, so there must be a better reason for inventing I/O completion ports. Their most important property is the controllable concurrency they provide. An I/O completion port's concurrency value is specified when it is created. This value limits the number of runnable threads associated with the port. When a thread waits on a completion port, the kernel associates it with that port. The kernel tries to prevent the total number of runnable threads associated with a completion port from exceeding the port's concurrency value. It does this by blocking threads waiting on an I/O completion port until the total number of runnable threads associated with the port drops below its concurrency value. As a result, when a thread calls **GetQueuedCompletionStatus**, it only returns when completed I/O is available, and the number of runnable threads associated with the completion port is less than the port's concurrency. The kernel dynamically tracks the completion port's runnable threads. When one of these threads blocks, the kernel checks to see if it can awaken a thread waiting on the completion port to take its place. This throttling effectively prevents the system from becoming swamped with too many runnable threads. Because there is one central synchronization point for all the I/O, a small pool of worker threads can service many clients.

Unlike the other Win32 synchronization objects, threads that block on an I/O completion port (by using **GetQueuedCompletionStatus**) unblock in last in, first out (LIFO) order. Because it does not matter which thread services an I/O completion, it makes good sense to wake the most recently active thread. Threads at the bottom of the stack have been waiting for a long time, and will usually continue to wait, allowing the system to swap most of their threads' memory resources out to disk. Threads near the top of the stack are more likely to have run recently, so their memory resources will not be swapped to disk or flushed from a processor's cache. The net result is that the number of threads waiting on the I/O completion port is not very important. If more threads block on the port than are needed, the unused threads simply remain blocked. The system will be able to reclaim most of their resources, but the threads will remain available if there are enough outstanding transactions to require their use. A dozen threads can easily service a large set of clients, although this will vary depending on how often each transaction needs to wait. Note that the LIFO policy only applies to threads that block on the I/O completion port. The completion port delivers completed I/O in first in, first out (FIFO) order. See the figure below.

{ewc msdncd, EWGraphic, bsd23458 0 /a "SDKV.WPG"}

Tuning the I/O completion port's concurrency is a little more complicated. The best value to pick is usually one thread per CPU. This is the default if zero is specified at the creation of the I/O completion port. There are a few cases where a larger concurrency is desirable. For example, if your transaction requires a lengthy computation that will rarely block, a larger concurrency value will allow more threads to run. The

kernel will preemptively timeslice among the running threads, so each transaction will take longer to complete. However, more transactions will be processing at the same time, rather than sitting in the I/O completion port's queue, waiting for a running thread to complete. Simultaneously processing more transactions allows your application to have more concurrent outstanding I/O, resulting in higher use of the available I/O throughput. It is easy to experiment with different values for the I/O completion port's concurrency and see their effect on your application.

The standard way to use I/O completion ports in a server application is to create one handle for each client [by using **ConnectNamedPipe** or **listen**, depending on the interprocess communication (IPC) mechanism], and then call **CreateIoCompletionPort** once for each handle. The first call to **CreateIoCompletionPort** will create the port. Subsequent calls associate additional handles with the port. After a client establishes a connection and the handles are associated with the I/O completion port, the server application posts an overlapped read to the client's handle. When the client writes a request to the server, this read completes and the I/O system queues an I/O completion packet to the completion port. If the current number of runnable threads for the port is less than the port's concurrency, the port will become signaled. If there are threads waiting on the port, the kernel will wake up the last thread (remember, waits on I/O completion ports are satisfied in LIFO order) and hand it the I/O completion packet. When there are no threads currently waiting on the port, the packet is handed to the next thread that calls **GetQueuedCompletionStatus**. The Windows NT 3.5 Software Development Kit contains source code for SOCKSRV, a simple network server that demonstrates this technique.

The most efficient scenario occurs when there are I/O completion packets waiting in the queue, but no waits can be satisfied because the port has reached its concurrency limit. In this case, when a running thread completes a transaction, it calls **GetQueuedCompletionStatus** to pick up its next transaction and immediately picks up the queued I/O packet. The running thread never blocks, the blocked threads never run, and no context switches occur. This demonstrates one of the most interesting properties of I/O completion ports—the heavier the load on the system, the more efficient they are. In the ideal case, the worker threads never block, and I/O completes to the queue at the same rate that threads remove it. There is always work on the queue, but no context switches ever need to occur. After a thread completes one transaction, it simply picks the next one off the completion port and keeps going.

Occasionally, an application thread may need to issue a synchronous read or write to a handle associated with an I/O completion port. For example, a network server may get partially through one transaction before discovering it needs more data from the client. A normal read would signal the I/O completion port, causing a different thread to pick up the I/O completion and process the remainder of the transaction. For this reason, Win32 extends the semantics of **ReadFile** and **WriteFile** to allow an application to override the I/O completion port mechanism on a per-I/O basis. The application makes a normal overlapped call to **ReadFile** or **WriteFile** with an **OVERLAPPED** structure that contains a valid hEvent handle. To distinguish this call from the normal case, the application also sets the low bit of the hEvent handle. Because Win32 reserves the low two bits of a handle, **ReadFile** and **WriteFile** use the low bit as a "magic bit" to indicate that this particular I/O should not complete to the I/O completion port. Instead, the application uses the normal Win32 overlapped completion mechanism (wait on hEvent, or call **GetOverlappedResult** with fWait==TRUE).

Windows NT 3.51 adds one more application programming interface (API) for dealing with I/O completion ports. **PostQueuedCompletionStatus** lets an application queue its own special-purpose packets to the I/O completion port without issuing any I/O requests. This is useful for notifying worker threads of external events. For example, a clean shutdown might require each thread waiting on the I/O completion port to perform thread-specific cleanup before calling **ExitThread**. Posting one application-defined "cleanup and shutdown" packet for each worker thread notifies each worker thread to clean up and exit. Because the caller of **PostQueuedCompletionStatus** has complete control over all the return values of **GetQueuedCompletionStatus**, an application can define its own protocol for recognizing and handling these packets.

4. Synchronization Primitives

Win32 provides a wide assortment of synchronization objects. These objects include events (both auto-reset and manual-reset), mutexes, semaphores, critical sections, and even raw interlocked operations. For simply protecting access to a data structure, all these objects will work fine. But in some circumstances, tthey can vary dramatically in performance. Any efficient server application must understand the tradeoffs inherent in each synchronization object.

4.1 Mutexes, Events, and Semaphores

Mutexes, events, and semaphores are all powerful synchronization objects provided directly by the Windows NT kernel. Because they are real Win32 objects, they are inheritable, have security descriptors, and can be named and used to synchronize multiple processes. **WaitForMultipleObjects** (and **MsgWaitForMultipleObjects**) provide plenty of power and flexibility when combining multiple synchronization objects of this type.

The kernel directly manages the synchronization of these objects, and will perform an immediate context switch when a thread blocks on one of these objects. Because accessing synchronization objects requires a kernel call, some overhead is involved. In cases where the synchronization period is very short and very frequent, this overhead (and any resulting context switches) may be much greater than the synchronization period.

4.2 Critical Sections

Unlike the flexible kernel synchronization objects, a Win32 critical section does only one thing. A critical section is a very fast method for mutual-exclusion within a single multithreaded process. **EnterCriticalSection** grants exclusive ownership of a critical section object, and **LeaveCriticalSection** releases it. Because critical sections are not handle-based objects, they cannot be named, secured, or shared across multiple processes. They also cannot be used with **WaitForMultipleObjects** or even **WaitForSingleObject**. This simplicity allows them to be very fast at mutual-exclusion. When there is no contention for a critical section, only a few instructions are needed to acquire or release it. When contention for a critical section occurs, a kernel synchronization object is automatically used to allow threads to wait for and release the critical section. As a result, critical sections are usually the fastest mechanism for protecting critical code or data. The critical section only calls the kernel to context switch when there is contention and a thread must either wait or awaken a waiting thread.

4.3 Spinlocks

In rare cases, it may be necessary to build your own synchronization mechanism. On an SMP system, normal memory references are not atomic. If two processors are simultaneously modifying the same memory location, one of the processor's updates will be lost. Performing atomic memory updates requires special processor instructions. X86 architectures provide the LOCK prefix to exclusively lock the memory bus for the duration of an instruction. RISC architectures (such as Mips, Alpha and PowerPC) provide a load-linked/store-conditional sequence of instructions for atomic updates. There are three Win32 APIs—**InterlockedIncrement**, **InterlockedDecrement**, and **InterlockedExchange**—that use these instructions to perform atomic memory references in a portable fashion. They can be used to implement spinlocks or reference counts without relying on the Win32 synchronization primitives. Do not confuse application-level spinlocks with the kernel spinlocks used internally by the Windows NT executive and I/O drivers.

The performance of the interlocked routines varies greatly, depending on the underlying hardware. Processor architecture, memory bus design, and cache effects all have a big impact on how fast the hardware can perform interlocked operations. One way to implement a spinlock is to use a value of zero to represent a free spinlock. When a thread needs to acquire the spinlock, it uses **InterlockedExchange** to set its value to 1. The spinlock is acquired if the result of the **InterlockedExchange** is 0, otherwise the attempt has failed and must be retried. There are many different strategies for retrying the lock acquisition (or, "spinning"). The best method depends on many factors. The hardware, memory cache policy, frequency of lock acquisition, and length of time the lock is held all make a difference. An important point to remember when selecting a retry policy is that the interlocked routines can be very expensive in their use of the system's memory bus. Spinning in a loop of **InterlockedExchange** calls is a good way to reduce the available memory bandwidth and slow down the rest of the system. It is better to read the lock's value in a loop, and only retry the **InterlockedExchange** when the lock appears free.

Spinlocks are a very efficient method of synchronizing small sections of code, but they do have some serious drawbacks. If the thread that owns the spinlock blocks for any reason, (to wait for I/O or a page fault, for example) all the threads in the application system must spin on the lock until the owning thread completes its wait and releases the lock. Because the Windows NT kernel cannot tell the difference between spinning on a spinlock and doing useful work, threads that are doing nothing but spinning will waste valuable CPU time. Even if the thread does not block, the kernel may summarily preempt it when it uses up its quantum or a higher priority thread becomes runnable. Again, because the Windows NT kernel does not know when a thread owns a spinlock, there is no way it can avoid preempting threads before they have a chance to release the lock. A Win32 synchronization object, on the other hand, immediately context switches to another runnable thread instead of wasting time spinning.

Boosting a thread's priority to real time will protect it from being preempted by most of the other threads in the system, but this is a fairly drastic solution. If real-time threads use all the CPUs, the system will appear completely frozen. This makes it hard to debug your application or even terminate it if something goes wrong! Because of these risks, Windows NT controls access to real-time threads that use its security model. Only processes running in a user account with the right to increase scheduling priority may change their thread priority to one of the potentially dangerous levels. Even real-time priority is not a complete solution, because the kernel scheduler round-robins real-time threads at the same priority. If a thread acquires a spinlock, then exhausts its quantum, any other thread trying to acquire the lock must spin until the owning thread is rescheduled and releases the spinlock. To solve this, sophisticated algorithms for spinning can be developed that detect when the owning thread has spent too much time spinning and should yield to another thread. A thread can yield the remainder of its quantum by calling **Sleep** with a sleep time of zero milliseconds. When this occurs, a yielding thread goes to the end of its ready queue, and another thread of the same priority can be given the CPU. A backoff spin algorithm like this is difficult to tune correctly for complex applications. Determining how long a thread should spin before giving up and yielding is critically important. Unnecessary yielding negates any performance advantage spinlocks have over critical sections, while insufficient yielding can occasionally cause drastic performance degradation if a thread is preempted while owning a spinlock.

A spinlock's limitations are severe enough that they should not be considered unless your application cannot tolerate the overhead of blocking in the kernel when a lock is owned. For this reason, you should use critical sections instead.

5. Managing Memory Usage

Processor cycles are not the only resource managed by the operating system. Efficient use of physical memory is critical to performance. Windows NT balances the available physical memory between the system, the file cache, and the applications by using working sets. The working set of a process consists of the set of resident physical pages visible to the process. When a thread accesses a page that is not in the working set of its process, a page fault occurs. Before the thread can continue, the virtual memory manager must add the page to the working set of the process. A larger working set increases the probability that a page will be resident in memory, and decreases the rate of page faults. One of the most critical parameters for a file server is the size of the file cache's working set. So in order to maximize file server performance, a default Windows NT Server installation increases the file cache at the expense of applications.

On the other hand, an application server's performance depends more on the working set of the application than on the size of the file cache, so this parameter should be changed on an application server. To change this setting, choose the Network applet from Control Panel. In the Installed Network Software box, click Server, then click Configure. This brings up a dialog box that lets you change this parameter to favor application performance.

Windows NT tries to do a good job of sharing physical memory between the system and the application, but sometimes an application needs to reserve more memory resources than it would normally get. To allow applications to request this special treatment, Windows NT version 3.5 introduces two new Win32 APIs: **GetProcessWorkingSetSize** and **SetProcessWorkingSetSize**. As with real-time priority threads, increasing an application's working set requires great care in order to avoid unpleasant effects on the rest of the system. Also like real-time priority threads, the process must hold the privilege to increase scheduling priority in order to override the system's decisions with these APIs. Although the system will do its best to honor the working set limits, low-memory situations can cause a process's working set to drop below the minimum size. If an application needs to force certain pages of memory to remain resident, it must use **VirtualLock**.

6. Caches

Computers that run Windows NT generally have a fast memory cache between the CPU and main memory. This takes advantage of memory access locality to allow most of the CPU's memory references to complete at the speed of the fast cache memory, instead of the much slower speed of main memory. Without this cache, the slower speed of DRAM memory would cripple the performance of modern, high-speed processors. In SMP systems, cache memory has an additional function that is vital to system performance. Each processor's memory cache also insulates the main shared memory bus from the full memory bandwidth demand of the combined processors. Any memory access that the cache can satisfy will not need to burden the shared memory bus. This leaves more bandwidth available for the other processors.

Any system that uses caches depends on the locality of memory accesses for good performance. SMP systems that provide separate caches for each processor introduce additional issues that affect application performance. Memory caches must maintain a consistent view of memory for all processors. This is accomplished by dividing memory into small chunks (that make up a *cache line*) and by tracking the state of each chunk present in one of the caches. To update a cache line, a processor must first gain exclusive access to it by invalidating all other copies in other processors' caches. When the processor has exclusive access to the cache line, it may safely update it. If the same cache line is continuously updated from many different processors, that cache line will bounce from one processor's cache to another. Because the processor cannot complete the write instruction until its cache acquires exclusive access to the cache line, it must stall. This behavior is called *cache sloshing*, because the cache line "sloshes" from one processor's cache to another.

One common cause of cache sloshing is when multiple threads continuously update global counters. You can easily fix these counters by keeping separate variables for each thread, then summoning them when required. A more subtle variant of the problem occurs when two or more variables occupy the same cache line. Updating any of the variables requires exclusive ownership of the cache line. Two processors updating different variables will slosh the cache line as much as if they were updating the same variable. You can be remedy this by simply padding data structures to ensure that frequently accessed variables do not share a cache line with anything else. Packing variables that are frequently accessed together into a single cache line can also improve performance by reducing the traffic on the memory bus. Most current systems have 32-byte cache lines, although cache lines of 64 bytes or more will show up in future systems.

Cache sloshing can be simple to fix, but very difficult to find. A profiling tool that tracks the total time spent in different functions is helpful, but plenty of guesswork and intuition is still necessary. Compare profiles of your program running on configurations with different numbers of processors. Any functions that take proportionally more time as the number of processors increases are likely victims of cache sloshing. As more processors compete for the same cache lines, the instructions that access those cache lines will run slower and slower. The function will not actually execute more instructions, but each instruction that needs to wait for the cache will take longer to complete, thus increasing the total time spent in the function.

7. Conclusion

After carefully designing and tuning your application, you may find that it still does not scale well. Sometimes, it is not only the software that is responsible. The availability of Windows NT has inspired an explosion of SMP computer designs. These computers range across a broad spectrum from personal workstations to million-dollar superservers. Building an SMP system presents even more design tradeoffs than building a uniprocessor system. As a result, many SMP computers vary widely in their overall performance and scalability. Because no industry standard benchmark for these computers has emerged yet, comparisons of different platforms can be difficult. Of course, existing benchmarks can test common components such as the disk or video subsystem. However, the memory bus, one of the most critical performance parameters for an SMP computer, can easily become a bottleneck for SMP applications.

While most computers scale well when the application runs mainly in the memory cache, many applications require working sets that are much larger than the cache. When multiple processors are continually contending for access to the main memory bus, the total main memory bandwidth is very important. The listing below contains the source code for MEMBENCH, a short program that tests the raw memory throughput of SMP computers. MEMBENCH measures the time required for multiple threads to modify a large array. When each thread accesses the array sequentially, locality is high and scalability is very good. As the stride used to step through memory increases, the locality decreases, causing more cache misses and dramatically decreasing the overall throughput and scalability.

```
--*/

#include <windows.h>

#include <stdio.h>

#include <stdlib.h>


typedef struct _THREADPARAMS {

    DWORD ThreadIndex;

    PCHAR BufferStart;

    ULONG BufferLength;

    DWORD Stride;

} THREADPARAMS, *PTHREADPARAMS;




DWORD MemorySize = 64*1024*1024;

HANDLE StartEvent;

THREADPARAMS ThreadParams[32];

HANDLE ThreadHandle[32];

ULONG TotalIterations = 1;
```

```
DWORD WINAPI
MemoryTest(
    IN LPVOID lpThreadParameter
    );

main (argc, argv)
    int argc;
    char *argv[];
{
    DWORD CurrentRun;
    DWORD i;
    SYSTEM_INFO SystemInfo;
    PCHAR Memory;
    PCHAR ThreadMemory;
    DWORD ChunkSize;
    DWORD ThreadId;
    DWORD StartTime, EndTime;
    DWORD ThisTime, LastTime;
    DWORD IdealTime;
    LONG IdealImprovement;
    LONG ActualImprovement;
    DWORD StrideValues[] = {4, 16, 32, 4096, 8192, 0};
    LPDWORD Stride = StrideValues;
    BOOL Result;

    //
    // If you have an argument, use that as the number of iterations.
    //
    if (argc > 1) {
        TotalIterations = atoi(argv[1]);
```

```
    if (TotalIterations == 0) {
        fprintf(stderr, "Usage: %s [# iterations]\n",argv[0]);
        exit(1);
    }
    printf("%d iterations\n",TotalIterations);
}
//
// Determine how many processors are in the system.
//
GetSystemInfo(&SystemInfo);

//
// Create the start event.
//
StartEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
if (StartEvent == NULL) {
    fprintf(stderr, "CreateEvent failed, error %d\n",GetLastError());
    exit(1);
}

//
// Try to boost your working set size.
//
do {
    Result = SetProcessWorkingSetSize(GetCurrentProcess(), MemorySize,
             MemorySize*2);
    if (!Result) {
        MemorySize -= 10*1024*1024;
    }
} while ( !Result );
```

```
printf("MEMBNCH: Using %d MB array\n", MemorySize / (1024*1024));

//
// Allocate a big chunk of memory (64MB).
//
Memory = VirtualAlloc(NULL,
                      MemorySize,
                      MEM_COMMIT,
                      PAGE_READWRITE);
if (Memory==NULL) {
    fprintf(stderr, "VirtualAlloc failed, error %d\n",GetLastError());
    exit(1);
}

do {
    printf("STRIDE = %d\n", *Stride);
    for (CurrentRun=1; CurrentRun<=SystemInfo.dwNumberOfProcessors;
         CurrentRun++) {

        printf("  %d threads: ", CurrentRun);
        //
        // Start the threads, and let them party on the
        // memory buffer.
        //
        ResetEvent(StartEvent);

        ChunkSize = (MemorySize / CurrentRun) & ~7;

        for (i=0; i<CurrentRun; i++) {
            ThreadParams[i].ThreadIndex = i;
            ThreadParams[i].BufferStart = Memory + (i * ChunkSize);
```

```
            ThreadParams[i].BufferLength = ChunkSize;
            ThreadParams[i].Stride = *Stride;

            ThreadHandle[i] = CreateThread(NULL,
                                           0,
                                           MemoryTest,
                                           &ThreadParams[i],
                                           0,
                                           &ThreadId);
            if (ThreadHandle[i] == NULL) {
                fprintf(stderr, "CreateThread %d failed, %d\n", i,
                        GetLastError());
                exit(1);
            }
        }

        //
        // Touch all the pages.
        //
        ZeroMemory(Memory, MemorySize);

        //
        // Start the threads and wait for them to exit.
        //
        StartTime = GetTickCount();
        SetEvent(StartEvent);

        WaitForMultipleObjects(CurrentRun, ThreadHandle, TRUE,
INFINITE);
        EndTime = GetTickCount();
```

```
            ThisTime = EndTime-StartTime;

            printf("%7d ms",ThisTime);
            printf(" %.3f MB/sec",(float)(MemorySize*TotalIterations)/
                   (1024*1024) / ((float)ThisTime / 1000));

            if (CurrentRun > 1) {
                IdealTime = (LastTime * (CurrentRun-1)) / CurrentRun;
                IdealImprovement = LastTime - IdealTime;
                ActualImprovement = LastTime - ThisTime;
                printf("  (%3d %% )\n",(100*ActualImprovement)/
                       IdealImprovement);
            } else {
                printf("\n");
            }
            LastTime = ThisTime;

            for (i=0; i<CurrentRun; i++) {
                CloseHandle(ThreadHandle[i]);
            }
        }

        ++Stride;
    } while ( *Stride );
}

DWORD WINAPI
MemoryTest(
    IN LPVOID lpThreadParameter
    )
{
```

```
    PTHREADPARAMS Params = (PTHREADPARAMS)lpThreadParameter;
    ULONG i;
    ULONG j;
    DWORD *Buffer;
    ULONG Stride;
    ULONG Length;
    ULONG Iterations;

    Buffer = (DWORD *)Params->BufferStart;
    Stride = Params->Stride / sizeof(DWORD);
    Length = Params->BufferLength / sizeof(DWORD);
    WaitForSingleObject(StartEvent,INFINITE);

    for (Iterations=0; Iterations < TotalIterations; Iterations++) {
        for (j=0; j < Stride; j++) {

            for (i=0; i < Length-Stride; i += Stride) {

                Params->BufferStart[i+j] += 1;
            }
        }
    }
}
```

As with any application that needs tuning, developing a scalable application requires attention to small details and careful design. Windows NT provides an excellent framework for taking advantage of powerful SMP platforms, but it also provides features that are unfamiliar to most programmers. Understanding how to combine powerful features such as threads, asynchronous I/O, and completion ports is the key to unlocking the performance that Windows NT offers.

Windows NT Security in Theory and Practice

Ruediger R. Asche
Microsoft Developer Network Technology Group

May 9, 1995

Abstract

This article is first in a series of technical articles that describe the implementation and application of a C++ class hierarchy that encapsulates the Windows NT® security application programming interface (API). The series consists of the following articles:

"Windows NT Security in Theory and Practice" (introduction)

"The Guts of Security" (implementation of the security class hierarchy)

"Security Bits and Pieces" (architecture of the sample application suite)

"A Homegrown RPC Mechanism" (description of the remote communication implemented in the sample application suite)

In this article, I will discuss security coding on a rather high level; that is, I will show how security can manifest itself in server code without presenting the actual code. CLIAPP/SRVAPP, a sample application suite that consists of a database client and server, illustrates the concepts introduced in this article series.

Introduction

If you have a rather hazy notion of what security programming in Win32® is about, or if you are interested in security programming from a conceptual point of view, this article is for you. If you are already familiar with the concepts of security programming and would like to see working code, or you wish to plug the C++ security library I provide into your server application, you should skim the last section of this article to get an idea of what the sample application suite does, and then proceed with the next article in this series, "The Guts of Security.".

Security should be fairly straightforward to implement in an operating system, right? I mean, all it should take to assign a certain security level to an arbitrary object is a single function call, such as **GrantAccessTo** or **DenyAccessTo**, right?

Unfortunately, the Windows NT® security application programming interface (API) doesn't appear to be that straightforward. It includes a plethora of functions that relate to security, and already the task of, say, opening up an object to only one user is very complex.

To utilize the security API appropriately, you need to understand it at several levels:

- The first level is understanding the data structures: access control lists (ACLs), access control elements (ACEs), security descriptors (SDs), and security IDs (SIDs). The C++ classes I provide will allow you to exploit Windows NT security without working with these data structures directly. In the article "The Guts of Security," I explain how these C++ classes are implemented.

- The second level is understanding the semantics of ACLs (although you don't need to understand how they work). Depending on the order in which ACLs are built, access to the same user may or may not be granted. In the "Toying with Security" section later in this article, we will see how this ordering affects you.

- The third level is understanding how the operating system itself employs security. You can think of the security API as a set of services that are provided for server applications to protect objects from unauthorized access by clients, similar to the way the event log API provides services that help device drivers and applications log errors and notification events.

  As long as these services are used only by third-party applications, it is fairly easy to understand how security works. However, Windows NT is a secure operating system (to stick with our previous definition, the operating system uses the services provided by the security API), and furthermore, networks based on Windows NT also rely on security very heavily. Therefore, the way security is incorporated into the system itself is rather obscure. (I will examine this issue in a future article.)

Before I dive into the subject matter, let me make a few suggestions for further reading that supplements the discussion in this article. First, you should definitely read Robert Reichel's two-part article "Inside Windows NT Security," which appeared in the April 1993 and May 1993 issues of the *Windows/DOS Developer's Journal*. Robert is one of the key developers of the Windows NT security subsystem, and his discussion of the security components and data structures is probably the most comprehensive information you can find on the subject. Second, for a conceptual discussion of security and how it fits into the architecture of a network based on Windows NT, you should read the *Resource Guide* in the Windows NT version 3.1 Resource Kit.

Who Needs Security?

Before I go into any details, let me clarify why you might need security. You may not have to bother with security at all, unless the following holds true for you:

You are writing a server application, that is, an application that several users can access, and your application provides data structures that are restricted to only a subset of those users.

Note that this is a fairly broad definition, and deliberately so. Here are a few examples of applications that fit this category.

For stand-alone computers (that is, machines that are not tied into a network), you can write a service that starts up as Windows NT boots and keeps running even as multiple users log on and off the same machine. The service could provide information that is visible only to a few users—for example, if you wish to compile usage patterns or logon data, you might want to restrict access to that data to the machine's administrator(s).

A number of privileges are restricted on the system level. For example, the system Registry is protected so that only users with special privileges are allowed to add device drivers to the system. This is for security reasons—for example, a malicious user could misuse a device driver's ability to monitor user input to spy on other users' work. Security can also help stabilize your system. Consider a poorly written device driver installed by an unauthorized user. Such a driver could crash the machine while another user is working. By restricting the ability to register new device drivers to trusted users, we can shield a Windows NT machine from this kind of misuse.

A large number of server applications that work over a network as well as stand-alone will benefit from some kind of hook into the security system. For example, a database server might serve several users at the same time, some of whom may not be allowed to see some of the data in a given database. Let us assume that everybody in your company can query your employee database. Administrative personnel will need to access all the information on employees, whereas everyone else should be able to see only job titles and office numbers. If you restrict the database fields that contain information on salary and benefits to administrative personnel, you can, in effect, allow everyone in the company to use the same database without compromising security and privacy. We will look into this possibility later in this article.

A Microscopic View of Security

One of the problems with security is that there is nothing fancy or glitzy about applying the security API. Other people in my group write code that rotates teapots, displays animated images in a window, pops up cool new Windows® 95 controls, sends data back and forth through a MAPI channel, and so on. I always seem to pick the boring stuff. . .

As complex as security under Windows NT may appear, it is rather straightforward on a microscopic level. Each Windows NT domain (or domain group) keeps a database of users that the domain knows about. A user who wishes to work on a computer within a Windows NT domain must identify himself or herself using a user name and a password. As soon as the security system verifies the password against the user database, the user (and every process he or she starts) is associated with an *access token*, an internal data structure that identifies the user.

The first thing you must know about security under Windows NT is that it is user-centric; that is, each line of code that attempts to access a secured object must be associated with a particular user—a user who must identify himself or herself to the client machine using a password. Each security check is made against the user identification. It is *not* possible, for example, to write code that prevents Microsoft® Excel from accessing an object. You can secure an object against access from Joe Blow running Microsoft Excel, but if Carla Vip is allowed to access the object, she can do so using Microsoft Excel or any other application she pleases—as long as Carla identifies herself on the client machine, using a password that is known only to her.

The security API, complicated as it may appear, accomplishes only two things:

- Auditing: A log entry is generated every time a specified operation on a specified object is attempted.

- Restricting object access: A function that a client application calls may succeed, fail with return code Error 5 (access denied), or fail for another other reason, depending on how the server assigns privileges.

I am absolutely serious. That is all. Error 5. Access denied. Instead of that error message, the user may see a dialog box that reads something like: "You do not have the privilege to remove the eggs from the carton." Internally, the application that pops up this dialog box probably contains code along the following lines:

```
if (!RemoveEggsFromCarton() && GetLastError() == ACCESS_DENIED)
  AfxMessageBox("You do not have the privilege to remove the eggs
                from the carton");
```

Security Mechanisms

Windows NT uses two mechanisms that cause a failed access attempt to return Error 5: verification against rights and verification against privileges. A *right* pertains to an action on an object, such as the right to suspend a thread or the right to read from a file. Rights are always associated with a certain *object* and a known *user*. For example, the right to read from a file must be associated with a file (to which this right is applied) and with a user who does or doesn't have that right. Likewise, the right to suspend a thread is useless unless it is associated with a specific thread and a user.

Privileges are predefined rights that pertain to operations on the system. For example, there are privileges to debug applications, to back up and restore storage devices, and to load drivers. Privileges are centered around users, not objects.

To make the distinction between the two a little bit clearer, let's look at the data structures that implement rights and privileges: A right is specified in a data structure called an *access control list*, or ACL. An ACL is normally associated with an object. A user is represented by an access token. When a user tries to access a secured object, his or her access token is checked against the object's ACL. The access token contains the unique identifier (the security ID, or SID) that represents the user. Each right in an ACL is associated with a SID; this way, the security subsystem knows the rights associated with each user.

Privileges, on the other hand, are encoded in the access token, so no objects are associated with them. To determine whether a user is allowed to do something that is associated with a privilege, the security subsystem examines the access token.

Furthermore, whereas rights require the specification of an action (the right to do what?—for example, to read a file or to suspend a thread), privileges do not (that is, the user either does or does not hold them). The action that goes with the privilege is implied in the privilege itself.

The reason why privileges are encoded in the access token is that most privileges override security requirements. For example, a user who is allowed to back up a storage device must be able to bypass file security—adding a new ACE to every single file on a hard drive just to allow the user to touch the file is simply not feasible. Thus, the code to back up a storage device first checks to see whether the user attempting the backup has backup privileges; if so, individual file security is ignored.

The set of privileges that can be associated with an access token is hardcoded and cannot be extended by an application. Server applications can implement customized security regulations using specific rights and generic mappings.

There are two types of ACLs: discretionary (DACL) and system (SACL). DACLs regulate object access, and SACLs regulate auditing.

Controlling access

In most cases, Error 5 is generated internally by a Windows NT-specific Win32 function called **AccessCheck**. This function takes as input a user's access token, a desired privilege, and an ACL (this simplification is good enough for now; we'll look into details later). An ACL is basically a list of small data structures (called *access control elements*, or ACEs), each of which specifies one user or a group of users, a set of rights, and the information on whether the rights are granted or denied. For example, an ACL might have an ACE that reads, "The right to remove eggs from the carton is explicitly denied to the users Elephant and Bozo," followed by an ACE that contains the entry, "The right to remove eggs from the carton is explicitly granted to Betty Crocker and all users in the CHEFS group."

ACLs are typically associated with objects and can be built dynamically from your server application. For example, if a file object is associated with an ACL, whenever an application tries to open that file object, the ACL will be consulted to determine whether the user who is running the application is allowed to open the file.

The **AccessCheck** function is called internally from a number of system functions, for example, **CreateFile** (when a user attempts to open a file on an NTFS partition or on a named pipe) and **OpenFileMapping**. However, a Win32 server application can call **AccessCheck** directly, thereby protecting any object it wishes to.

Note that functions of the security API are called *only* by server applications; clients never request or employ security directly. All that clients ever see of Windows NT security is Error 5. This allows Windows NT security to work regardless of which software the client is running. All that is required is the server's ability to identify the client in the domain's security database and translate any incoming request from the client to a function call on the server side. This function either calls **AccessCheck** implicitly, or its result may or may not be sent, depending on the outcome of **AccessCheck** on the server side. (This sounds very abstract, but I do exactly that in the server application later on.)

Part of the confusion in Windows NT security is the fact that calls to **AccessCheck** can be very obscure. For example, the ability of Windows NT to monitor attempts to install device drivers is a very shady notion—which "object" does a user try to access when attempting to add a device driver? Where exactly does the system call **AccessCheck** and display an error to the user if necessary?

In the case of device drivers, the answer is not too difficult: Because device drivers and the system interact through the Registry (Windows NT loads device drivers by traversing a Registry subtree, interpreting each entry, and trying to execute the driver binaries that are specified in the individual Registry keys), the objects that Windows NT protects are Registry keys, which are securable objects under Windows NT. On the Win32 API level, any attempt to manipulate the Registry will be translated into one of the functions that work on the Registry, such as **RegOpenKey**, which calls **AccessCheck** internally.

Note that aside from the Registry protection, there is also a security issue with the driver binaries. A frustrated hacker who is denied access to the Registry could still replace an existing driver executable file with a file that clones the existing driver and also has additional functionality—this process does not require access to the Registry, so how can Windows NT prevent this kind of misuse? Rather easily, by requiring that the driver binaries reside on an NTFS partition, and by restricting access to the binaries. This way, an attempt to replace the driver binary (which will inevitably end in a **DeleteFile** or **CreateFile** call on the Win32 API level) will be trapped by **AccessCheck**, and our malicious hacker will be out of luck.

Other system-provided, secured objects may be more difficult to figure out. For example, what exactly prevents a user from accessing a protected network share? What refuses to let you open the service control manager on a remote machine? What is it on the system level that makes Windows NT airtight? Or, even trickier, what makes some of the security functions themselves fail with Error 5, access denied? Imagine what would happen if an application could freely manipulate its access token or call a security function to change the privileges on objects it can see. In that case, it would be easy to bypass security

by simply adjusting the entries in the ACLs and tokens. Thus, there must be some kind of "meta-security"; that is, a mechanism to protect the security features themselves from misuse. How is that implemented?

Tune in again next week for our next episode. . . I will discuss the secure architecture of Windows NT itself in a future article. This article and its siblings, "The Guts of Security" and "Security Bits and Pieces," deal only with the easy issue: how to secure your objects in your server application. Back to our main program after these exciting messages from our sponsor.

Note that one of the consequences of a security implementation based on **AccessCheck** is that security relies heavily on architectures that allow only well-known entry points to secured objects. For example, the Windows version 3.1 family of operating systems includes a large number of different entry points to the file system: int 21h (which interacts with the file system), int 13h (which interacts with the disk device driver), and several types of C run-time and Windows API functions (such as **OpenFile** and **_fopen**) that provide file system access. It would make no sense from a security point of view to call a function such as **AccessCheck** in the internal implementation of **OpenFile**, when an application could simply call **_fopen** and bypass file security. As long as all variations of file-open calls are translated into one "secured" call, we are fine; but as soon as one variation performs security checks and another doesn't, we have a security problem.

As a side note, this "open file system" architecture in the 16-bit Windows system is one of the major headaches for vendors who provide security add-ons, such as encryption software and hardware.

When you write a secured server application, it is absolutely necessary that you design your application to be airtight; that is, you must protect *all* means by which clients may access your sensitive data. In the sample application, I show how remote access from a client to a server may be protected, but if the client and server happen to reside on the same machine, and the shared memory in which the database resides is not protected, the client can "sneak into" the database, thus violating security. One of the challenges of a secured system is to make the sensitive data airtight—this may be a fairly intricate task, as we saw in the case where protecting Registry entries alone is not good enough to protect a machine's device drivers.

Access Right Types

With the security API, the system can help you regulate access to almost any kind of object. But what does "access" mean? Isn't the type of access you imply when you talk about database fields something completely different from accessing, say, the message loop of another window?

Exactly, and that is why "access" is a fairly generic term in the security API. Instead of hard-coding access types such as "the right to open, close, read from, or write to an object," access in Windows NT is defined as a collection of bits in a mask. The security subsystem matches the bits in the user's access mask with the bits in the object's access mask. This enables us to design an employee database, for example, in such a way that administrators can read and write information on payroll and benefits; managers can read, but cannot write to, these database fields; and nobody else has read or write access.

By the same token, your application can define its very own access types. For example, if your application wants to secure an OpenGL™ object that can be shared (in the sense that several users could call functions that manipulate the object on the screen), you could simply define unique access rights for all the cool things you can do with OpenGL objects (for example, rotate, stretch, flip, and remove), and assign each user who works on the image a unique subset of those rights.

The security API can work with three groups of rights:

- *Standard rights* (rights that provide the same operation for each object type).

- *Specific rights* (rights that mean something specific for each object type—objects of two different types may have the same bits in the rights mask set, but different interpretations of what those rights mean).

- *Generic rights*, which are roughly placeholders (rights such as GENERIC_READ and GENERIC_WRITE, which are applicable to almost all object types, but mean different things for different object types). Generic rights are mapped to standard and specific rights. This mechanism allows a server to establish a notion of "writing" and "reading" an object without really defining these actions. For example, for a database object in the CLIAPP/SRVAPP sample application suite, "reading" the database means "retrieving records," and "writing" means "adding or deleting records." A server application can work with generic rights, regardless of whether reading and writing refers to a file or to a database object, and the objects themselves can determine how the generic rights translate into specific rights.

So far, this has been a really abstract discussion. Let's look at some hands-on examples to clarify what we talked about.

Toying with Security

I argued earlier that servers are the only applications that need to call security API functions, so they can regulate access by client applications. Thus, a sample that demonstrates security requires at least two parts: a server application that allows or disallows a client to remove eggs from the carton, and a client application that attempts to remove the eggs.

Consequently, the sample code that I've provided with this article consists of two parts: a server application (SRVAPP.EXE) and a client (CLIAPP.EXE). To minimize the potential harm, my sample application suite does not work on eggs, but on data structures in memory. Let's see what you need to run these applications.

Setting Up Your Hardware

To run the sample code, you need at least one computer running Windows NT. This will be the server machine. The server machine will contain the data to be opened or denied to users.

This data will be accessed by applications running on a client machine. The client machine can be the same as the server machine, or it can be another machine running either Windows NT or any other operating system that can execute Win32 binaries and log onto a Windows NT domain (for example, Windows for Workgroups 3.11 with Win32s® extensions, or Windows 95). Please see the next section, "The Secret of Logging On," for more information on what happens when a user identifies himself or herself.

The server machine must be able to access security information for the user who logs onto the client machine. If you set up the client and server on the same machine, this requirement is automatically met. If the client is on a different machine, you must log onto the client machine as a user on a domain that the server can access.

Because security is based on users, you should first create a few test user accounts under which you can log onto the client machine. (On a Windows NT Server machine, you can use the User Manager for Domains application to create these accounts. On a Windows NT Workstation machine, the "normal" User Manager will do, as long as you have administrative privileges on the machine.) You should also create a few groups (also with the User Manager), and set up a few assignments between users and groups. For my test purposes, I created two groups, OOZLES and WABBOTS, and four users, Lillo, Gnorps, Alf, and Picard. I assigned users to groups as follows (we will come back to this scenario later on):

- Lillo is a member of WABBOTS, but not OOZLES.
- Gnorps is a member of OOZLES, but not WABBOTS.
- Alf is a member of both OOZLES and WABBOTS.
- Picard is not a member of either WABBOTS or OOZLES.

In the Windows NT security model, a user is identified by two components: the user name and a domain name. To create a user account on a specific domain, you must have administrative privileges on that domain. You will *not* need administrative privileges to run the tests, but only to create the user accounts and groups.

If you do not have the appropriate privileges to create new accounts in your domain, you can run both the server and the client applications on machines that are on the same network (for example, your corporate network), using users and user groups that already exist in your network's domain.

As I mentioned before, the client application does not have anything to do with security. It simply tries to access shareable objects that the server owns; all the code that manages security resides in the server. All the client will notice in terms of security is that it may be denied certain operations on the shared objects—some API functions that the client calls may simple return the Error 5 (access denied) instead of succeeding.

The Secret of Logging On

Now let me clarify what "logging on as" means. If your server machine is running the Windows NT Server operating system, you can use the User Manager for Domains application from the Administrative Tools Group to create user accounts. You can then configure the client machine to belong to the domain that Windows NT Server administers. A user who wants to log onto the client machine must identify himself or herself through an account that the server machine administers.

However, if the server is running the Windows NT Workstation operating system, a user on the client machine cannot use an account on the server. Note that if the user who is logged onto the server machine has administrative privileges, he or she can maintain user accounts on that local machine—in that respect, Windows NT is a stand-alone domain that follows the same security rules as a multi-machine domain. The only difference is that client machines cannot register themselves as belonging to the domain defined by a workstation.

However, you can still run the sample application suite, even if you are not running Windows NT Server. Do the following:

- Designate a Windows NT Workstation machine as the server machine. On this machine, log on as an administrator and create a few accounts (as we discussed in the last section).

- Put copies of CLIAPP.EXE and SRVAPP.EXE in a shared directory on the server machine.

- From the client machine (which, as we discussed earlier, does not need to run Windows NT), log onto the server using the following LAN Manager command:

  net use *drive*: *server**share* /user:*server**account* *

  where *drive* is the drive letter you wish to use for the connection, *server* is the name of the server machine, *share* is the share that contains CLIAPP.EXE, and *account* is an account on the server.

  For example, if the server machine is called SLACKER, and you create the above user accounts on that machine's domain, the logon would look something like this on the client side:

  net use K: \\slacker\database /user:slacker\lillo *

Using the password that the server machine assigned to the user account, you can now log onto the server machine and run the server application remotely. The server will now treat the client machine as if it were the user for whom we created an account. This is possible through a process known as *impersonation*: When the client connects to the share where CLIAPP.EXE resides, the server treats this connection as if the specified user had logged onto the server.

Using the Sample Application Suite

The sample application suite, which consists of a server application and a client application, demonstrates how to secure named pipes, mutexes, file mappings, and private objects.

Server side

The server application is a little database server. Bring up SRVAPP.EXE on the server machine and experiment with the commands in the Database menu: You can insert records (in this case, a record simply consists of two integer values), view the contents of the database, and remove records. No magic to that whatsoever. The main purpose of the server application is to provide a shell that demonstrates how to regulate access to the database from a client application—that is, the server accepts database requests from remote clients, decides whether the access is allowed (according to security requirements that the user of the server application specifies), and, depending on the outcome, grants or denies the database requests to the client.

Note that the database software is homegrown: that is, I supply all the logic and code to maintain the database. In this age of automation and powerful database controls, it makes little sense to reinvent the wheel. Fortunately, the encapsulation mechanisms of C++ allow easy replacement of the database, so for future enhancements of the server application, I plan to replace the homegrown database with an existing database control (for example, an OLE automation object provided by a database server application).

You can use two techniques to allow a client application to utilize the server's database:

- Let the client and server access the database locally, via shared memory. (This technique works only if the client and server applications are executed on the same machine.)

- Let the client and server access the database remotely, through network access. To use this technique, select the Wait to Connect command from the server's File menu.

See the third article in this series, "Security Bits and Pieces," for more information on these techniques.

Client side

On the client machine, follow these steps:

1. Log onto the client machine, using one of the test accounts that you created (to stick with the above sample, you could choose Lillo). Make sure that the server has been prepared to wait for incoming connections (you should have selected Wait to Connect from the server's File menu), then choose Connect to Server from the client's Remote Access menu, and specify the name of the server machine in the dialog box.

   Note Here lurks an opportunity for confusion. I mentioned earlier that security is user-based, but a named pipe is identified by the name of the *machine* on which the pipe was created. Thus, if the name of the server machine is SLACKER and you log onto the server machine as Gnorps, all security checks are performed based on the identification of Gnorps, but a named pipe is created using the name SLACKER. (Please see the "Garden Hoses at Work" article in the Development Library for more information on named pipes.)

   The client should now display a line in its main window saying that the named pipe could not be opened due to error—"access denied." Hah! We got 'im!

2. Now go back to the server application, choose Cancel Wait from the File menu, choose Named Pipe from the Permissions menu, and enter the following information in the Named Pipe Security dialog box:

   - Set User to "Lillo" or whatever account you used to log onto the client application.
   - Leave Domain blank for the time being. (This option defaults to the domain that you used to log onto the server machine.)
   - Choose Grant Access as the access type. This will grant Lillo access to the named pipe. Note that in my sample application suite, "grant access" means "grant the user access to *all* operations on the pipe," although you can change the code to selectively grant or deny read or write access to the pipe (we will see later on how this can be accomplished).

3. Now choose Wait to Connect again from the server's File menu, and select Connect to Server from the client's Remote Access menu, again choosing the name of the server machine. You should now see a message from the client application, saying that the connection has been established. Here we see the first application of security: granting a user access to a named-pipe object.

4. Now choose Disconnect from Server from the client's Remote Access menu. The server should now display the message, "Client terminated connection."

5. Bring up the Named Pipe Security dialog box from the server application's Permissions menu again, and deny access to WABBOTS. Choose Wait to Connect from the server's File menu again, and try to reestablish the connection as before. You should now receive the error message, "Could not open pipe—access denied" in the client's window again. Makes sense, doesn't it? After all, Lillo is a member of WABBOTS, so denying WABBOTS access to the named pipe should also block Lillo.

   Note You can change the permissions on the pipe while the pipe is connected, but if you do that, you will not see any change in behavior until the pipe is disconnected and reconnected to a

new client. The security system performs the access check whenever a client attempts to open the pipe. If the opening is successful, the access rights do not change until the pipe disconnects. For example, if you decide that a certain user (who currently has read/write permissions on the pipe) should only have read permissions on the pipe, you can change the permissions whenever you like. However, as long as that user has successfully opened the client end of the pipe for reading and writing, he or she can read from, and write to, the pipe until the pipe is disconnected. That is why you should have the client disconnect every time before you make security changes.

Take some time to play with the rights to your heart's delight—the Permissions dialog box lets you grant, deny, or revoke previously granted or denied rights. Try to get a feeling for how group rights relate to user rights; for example, what happens if access is denied to Lillo but granted to WABBOTS when the client is logged on as Alf? What happens if you have a more complex hierarchy of user groups (say, a three-level hierarchy) in which individual users are excluded from a high-level group, but included in a lower-level group? (For example, if both OOZLES and WABBOTS belong to the BALLOONS hyper-group, what if Lillo is in BALLOONS and WABBOTS, but not in OOZLES?)

What happens in the case of ambiguous rights? To test this, log off the client machine and log on as Alf, who, as you may recall, is a member of both OOZLES and WABBOTS. Grant WABBOTS access to the named pipe, and deny access to OOZLES. When the client tries to access the named pipe now, it should see the dreaded "access denied" message.

Does that mean that denying access is stronger than granting access? After all, as a member of both WABBOTS and OOZLES, Alf is both granted and denied access, so why is the client denied access when it tries to access the named pipe? Even stranger, if we now explicitly grant access to Alf, the client will still be denied access! How come?

There is a reason for everything. When Alf tries to connect to the named pipe, Alf's identity is matched against all access rights in order, and the first right that contains Alf is applied. The code that I provide deliberately adds all access-deny rights to the beginning of the list. When the code that tries to open the named pipe traverses the list of assigned rights, the first access that applies to Alf is the one that reads "access denied to OOZLES." As soon as the security system encounters this entry, the security check function returns immediately, regardless of what follows the entry in the security list. If the entry "access granted to WABBOTS" appeared before the "access denied to OOZLES" entry, the security system would return with an access grant.

So why does my application code stuff all of the denied rights in front of the granted rights? Remember that for each object that is associated with a set of rights, a user who has not explicitly been granted access is automatically denied it. Thus, we could simply grant access to the users who we believe should access the object, and let everybody else die in the default case. So why do we need access-denied entries at all?

Very simple: to refine rights we have granted before. Let us assume that we want everyone in the OOZLES group *except* Alf to be able to access the named pipe. Because OOZLES is potentially a fairly large group, we do not want to enumerate all users within OOZLES and grant each one (except Alf) access. An easier way to exclude Alf would be to deny access to Alf explicitly while granting access to OOZLES. That is where it makes perfect sense to put all access-denied elements in front of the list: While traversing the list of rights, the system would find Alf's denied entry first and return, whereas for every other user in OOZLES, the system would scan the list until it found the access-granted element.

More Security. . .

When you have tired of toying with this kind of thing, you are ready for the next step. Go back to a security assignment in which the client can successfully open a connection with the server. On the server side, add a few records to the database, then select View Contents from the server's Database menu. You should now see a few entries in the server's menu reading something like, "Element *x* has values *y* and *z*."

Now do the same thing on the client side: Select View Contents from the client's Remote Access menu. You should now see a number of messages on the server side reading, "Remote retrieve succeeded," and on the client side, you should see the same messages you saw earlier on the server side: "Element *x* has values *y* and *z*."

When I got to this point in my application design, I was pretty happy: I had designed a complete little RPC-based database server and client application in which the database could be accessed from both sides over a network. However, no security was yet involved (except for the named-pipe protection), so I added what you can see next.

Try to add a record from the client, using the Add Record command from the Remote Access menu. On the server side, you will see, "Remote insert failed—propagating error Access Denied," and on the client side, you will very simply see the message, "Could not insert element—access denied."

Why is that? Simple: I have designed the application such that the client can always read from the database, but not write to it unless that privilege is explicitly granted. Both inserting and removing records is considered writing to the database, so the client can enumerate the contents of the database (read from it), but cannot add or remove contents (write to it).

You have probably already spotted the Database command in the server's Permissions menu. Now is the time to use it: Bring up the Permissions/Database dialog box and use it exactly as you used the Permissions/Named Pipe dialog box. Grant access to the user in whose context the client application runs, and voilà! The next time the client application attempts to add a record, you will see the message, "Remote insert succeeded!" on the server and client sides, and viewing the contents of the database on either side will give you the new record. The same thing works with record deletions.

Phew! Now we have not only a client-server database system, but a *secured* client-server database system, on which the server can restrict access to the database to known and trusted users! Doesn't that cover all we wanted to demonstrate?

Yes and no. I made a point earlier that security must be airtight, and there is one more thing you should do before turning off your computer.

Shut down the client application, and restart it on the server machine. Yes, you heard right: For this test run, we will execute both the client and server applications on the same machine. It should come as no surprise to you that the application suite will behave *exactly* as it did before—with one little exception: When you select Open Shared Database from the client's Local Access menu, you will see the error message, "Access denied." (Attempting this option when the client and server are executing on different machines returns the message, "The filename is incorrect.") We have seen that one before, right? And the last time we saw this, we modified the Permissions dialog boxes, so let's choose the Shared File command from the server's Permissions menu and grant access to the user under whose name we logged on.

Now selecting the Open Shared Database command should succeed. Miraculously, the Add a Record, Remove a Record, and View Contents commands in the client's Local Access menu are not only enabled, but invoking them always succeeds, regardless of the permissions! Whew! To confirm that something weird is going on, grant the current user access to the named pipe (just as you did when the client and server executed on different machines), open a connection, and verify that the client is now denied

remote access, but can party on the database via the shared file! Security violation? Anarchy?

Not really. This little experiment demonstrates that a server application must be very careful in restricting access to its sensitive data through all paths available to the client. In our little case, we are fine because the client cannot access the shared memory unless it is granted explicit permission. However, if the code had not secured the shared memory area, it would have given the client a "trap door" for accessing the data, although the "proper" (in this case remote) access to the database was appropriately protected. Making a server application airtight can be one of the major challenges in security design.

Summary

We have established a context for Windows NT security programming and illustrated the concepts with a "black box" sample application suite. You should now have a conceptual idea of how security programming for Windows NT works. I'm sure you are dying to figure out how the server was coded to incorporate all the security magic. (Yeah, right. . . As I write that, I look out the window and wish I had brought in my motorbike today.) If so, you should proceed with the article "The Guts of Security," which goes into all of the gory details of security programming.

Developing Transport-independent Applications Using the Windows Sockets Interface

Presented by: David Treadwell

David Treadwell has been active in Windows Sockets since its inception, acting as a coauthor of the specification and developing the Windows NT Windows Sockets interface.

Introduction

In today's heterogeneous networks, one thing is clear: closed, proprietary standards are unwelcome. As the business of networking computers has evolved over the last two decades, hardware and software providers have learned that cooperation can yield very rewarding results. The Windows Sockets effort represents an extremely usable open networking standard developed by over 20 cooperating vendors in the networking community.

The Windows Sockets API (Application Programming Interface) provides applications with an abstraction of the networking software below it. The present Windows Sockets specification, version 1.1, defines this abstraction for the TCP/IP, or Internet protocol family. For many, the TCP/IP protocols represent the greatest common denominator between the many distinct systems that make up today's enterprise networks. In fact, the TCP/IP protocols were developed in a manner similar to the Windows Sockets specification: open cooperation between many interested parties with different requirements. Windows Sockets doesn't stop at TCP/IP, however. The level of abstraction is complete enough to support other protocol families as well, for example: the Xerox Network System (XNS) protocols, Digital's Dec-net protocol, or Novell's IPX/SPX family. For many, the attraction to Windows Sockets is the ability to develop and/or run a Windows Sockets-compatible application over any vendor's Windows Sockets-compliant TCP/IP implementation, while providing the ability to move the application easily to other networking protocols.

Windows Sockets is implemented as a DLL (dynamic link library) provided by the vendor of the given network protocol software. A Windows Sockets developer leverages the APIs from both Windows Sockets as well as the Windows operating system itself to create a network-aware Windows application. The following diagram illustrates the basic building blocks used to create a Windows Sockets application. The areas shaded in grey are provided by the network protocol vendor.

{ewc msdncd, EWGraphic, bsd23459 0 /a "SDKD.BMP"}

The Basic Building Blocks of a Windows Sockets Application

The goal of this article is to offer developers a taste of this powerful new API. We assume that the reader is familiar with both networking basics as well as Windows programming. To keep things simple, we will focus our discussion on using Windows Sockets to develop network applications over the TCP/IP protocol family, but will include several tips on how other transport protocols are supported with Windows Sockets.

Today, over 30 application vendors have announced the development of commercial applications to Windows Sockets such as X Windows servers, terminal emulators, and email systems. Several commercial and public domain versions of Windows Sockets-compliant TCP/IP stacks are available, and both Windows NT and Chicago include Windows Sockets implementations which are capable of supporting transport protocols in additions to TCP/IP, such as IPX/SPX. Many corporate developers are standardizing on Windows Sockets for heterogeneous client-server application development under Microsoft Windows.

Whether you're developing client-server, peer-to-peer, or distributed network applications, Windows Sockets represents a standard networking API for Microsoft Windows which will allow you to develop flexible networking applications for TCP/IP and other networking protocols.

The Birth of Windows Sockets...

The Windows Sockets specification is the result of a cooperative effort among over 20 vendors in the TCP/IP community. The charter of the group was simple: *to design a binary-compatible API for the TCP/IP protocol family under Microsoft Windows, allowing for future support of additional transport protocols.* The effort, led by Martin Hall of JSB Corporation, was kicked off at a Birds of a Feather session at the Fall '91 Interop networking conference.

Infuriated by a lack of standardization, TCP/IP application vendors like JSB were forced to develop their applications to be aware of several divergent APIs. This allowed their applications to run over multiple vendors' TCP/IP implementation, making their products available to the widest possible audience. With over 10 different TCP/IP implementations on the market, many vendors created an *abstraction layer* to the network interface, creating a common denominator which could be supported by all of their target implementations. Their application was then developed to this proprietary abstraction layer. *Providers,* or code which glued the application to a specific vendor's TCP/IP implementation, were developed for each of the TCP/IP implementations which the application desired to support.

This approach was both costly and frustrating. Application vendors were continuously updating their provider modules as TCP/IP implementors modified or updated their libraries. Moreover, new implementations were springing up quickly, and it took time before the appropriate provider could be made available to customers. Application vendors found it difficult to maintain, test, and support the multiple providers. This caused TCP/IP implementors difficulty as well, especially if a critical third- party application didn't run over their implementation. Customers were forced to choose a TCP/IP implementation based on their application needs rather than the merit of vendors' transports.

It would appear that getting the developers of TCP/IP transports and applications to work this out would make a lot of sense. Martin Hall acted as the catalyst to get things going quickly. In fact, vendors were so motivated to straighten out the TCP/IP networking API confusion that in just nine months the Windows Sockets committee published the first version of the specification. The first anniversary of the effort was christened by several technology and interoperability demonstrations at Fall Interop '92. The message was clear: Windows Sockets was *real*.

Windows Sockets Architectures

How are Windows NT and Chicago able to support several transport protocols with a single API set? The answer lies in the carefully designed network architectures of these operating systems. Because an understanding of the underlying network architecture is always helpful and frequently mandatory in designing and implementing fast, robust networking applications, we'll briefly describe how Windows Sockets fits into these operating systems.

Windows Sockets in Windows NT

The key to transport-independent Windows Sockets support in Windows NT is a common kernel-mode transport interface called *Transport Device Interface*, or TDI for short. All of the networking components of Windows NT go through TDI to access a transport protocol's services. TDI abstracts key differences between protocols, such as the format of transport addresses, and provides common entry points for typical transport features like sending data.

{ewc msdncd, EWGraphic, bsd23459 1 /a "SDKD.BMP"}

Networking Architecture of Windows NT

All the kernel-mode networking components of Windows NT use the TDI interface to speak to the transport layers below them. This use of a common interface allows easy addition or removal of transport protocols, since each transport protocol is completely separated from the layers which use it; in fact, each transport is packaged as a separate driver file.

However, TDI is not the only kernel-mode transport interface available in Windows NT. There is also the *Streams* interface which is based on the AT&T SVR4 Streams environment. Streams is useful for porting existing transport protocols to NT quickly and easily. However, it does impose a performance overhead on all transactions, since there is a mapping layer between the TDI calls made by upper layers and the internal interfaces used by Streams. The TCP/IP and IPX/SPX transport protocols supplied with the original release of Windows NT existed in the Streams environment, but Daytona, the next release of Windows NT, will include native TDI implementations of these transport protocols for improved performance.

The use of TDI as the interface underneath Windows Sockets solves most of the issues with multiple transport support, but some additional issues do remain. For example, because each transport protocol uses a different address format, how could Windows Sockets know which addresses are broadcast addresses? How can the Windows Sockets DLL know which transport device name corresponds to a given type of socket? And how can transports supply their own unique socket options like TCP/IP's SO_DONTROUTE option?

The answer is the use of user-mode "helper DLLs" which the Windows Sockets DLL (wsock32.dll) uses for carefully defined functionality. Each of the TDI transport protocols exposed through Windows Sockets supplies one of these helper DLLs as well as placing information in NT's registry about where to find the helper DLLs and what sorts of sockets each supports. The Windows Sockets DLL then calls into the helper DLL's entry points to learn about the format of transport addresses, to process transport-specific socket options, and more.

{ewc msdncd, EWGraphic, bsd23459 2 /a "SDKD.BMP"}

Windows Sockets Architecture of Windows NT

In addition to 32-bit Windows Sockets applications supported through wsock32.dll, Windows NT also supports 16-bit Windows Sockets applications through the file winsock.dll. This file is composed of minimal entry points which call into NT's "WOW" (Windows-on-Windows) subsystem, which widens the parameters to 32-bits and calls into wsock32.dll for the actual networking support. Thus, winsock.dll acts as a "thunking layer" between 16-bit applications and the rest of the operating system, which is all 32-bit.

Windows Sockets in Chicago

Note The information in this section pertains to the next major release of Windows, code named Chicago. All of the information in this section is preliminary and subject to change.

Chicago also supports multiple transports under Windows Sockets, using the same names, wsock.dll and winsock.dll, for the system DLLs so that application binary compatibility is preserved. However, internally the architecture of Windows Sockets in Chicago is significantly different than NT's architecture.

{ewc msdncd, EWGraphic, bsd23459 3 /a "SDKD.BMP"}

Windows Sockets Architecture of Chicago

One of the key differences between the Chicago and NT Windows Sockets architectures is that there is no thunking involved for 16-bit applications. The Windows Sockets DLL is cross-compiled into two different files, WSOCK32.DLL and WINSOCK.DLL. Each of these speaks directly to the underlying VxDs (virtual device drivers) which provide Windows Sockets support, thereby eliminating an extra layer for 16-bit applications.

Next, Chicago has multiple kernel-mode transport interfaces, including both TDI and the ECB (event control block) interface. Therefore, the drivers which translate between Windows Sockets calls and the transport drivers are different for each transport. The WINSOCK.386 VxD points the user-mode DLLs at the appropriate kernel driver, then the DLLs access WSTCP.386 or WSNW.386, which in turn speak to their transports over each transport's interface.

The Sockets Paradigm

The Sockets paradigm was first introduced in Berkeley UNIX® (BSD) in the early 1980s. Initially designed as a local IPC (inter-process communication) mechanism, sockets evolved into a network IPC mechanism for the built-in TCP/IP protocol family. A *socket* simply defines a bidirectional endpoint for communication between processes. Bidirectional simply implies that Windows Sockets allow applications to transmit as well as receive data through these connections.

A socket has three primary components: the interface to which it is bound (specified by an IP address), the port number, or ID to which it will be sending or receiving data, and the type of socket (either stream or datagram). Typically, a server application *listens* on a well-known port over all installed network interfaces. On the other hand, a client generally initiates communication from a specific interface from any port that the system has available. The type of the socket (stream or datagram) depends entirely on the needs of the application. Windows Sockets is closely related to the Berkeley sockets model; many of the APIs are identical, or very close. In addition to the Berkeley-style functions, Windows Sockets offers a class of asynchronous extensions which facilitate the development of more "Windows-friendly" applications. These extensions will be discussed in detail later in this article.

Stream vs. Datagram Sockets

The Windows Sockets model offers service for both connection-oriented and connectionless protocols. In the TCP/IP protocol family, TCP provides a connection-oriented service, whereas UDP (user datagram protocol) offers connectionless service. In the sockets model, connection-oriented service is offered by *stream* sockets, connectionless service is provided by *datagram* sockets.

TCP is a reliable, connection-oriented protocol, used by applications which either plan to exchange large amounts of data at a time, or by applications which require reliability and sequencing. For example, FTP (file transfer protocol), a protocol which facilitates the binary or ASCII transfer of arbitrarily large files, represents an application written to TCP or stream sockets. In contrast, if an application is willing to manage its own sequencing or reliability, or is using the network for low-bandwidth iterative processing, UDP is often used. An application which keeps system clocks synchronized by periodically broadcasting its system time would probably be written to use UDP.

Network-byte Order

Since Windows Sockets applications can't possibly be aware of what type of remote computer system that they will be dealing with *a priori*, it is necessary to define a common data representation model for vital information. Sockets chose the big-endian model for the "on-the-wire" data representation, known as *network byte order*.

The Windows Sockets interface offers APIs to application programmers to do the necessary conversion between the local system representation (or host byte order) and network byte order. There is no harm in using these routines on systems which store data in big-endian natively, in fact, it's encouraged. By religiously using the byte-ordering APIs, your code can be used on systems with different internal representations without inheriting byte-ordering problems, thereby making your code more portable.

A Guided Tour of the Windows Sockets API

Although the Windows Sockets specification defines all of the Windows Sockets functions and structures, this guided tour of the API will give you a basic understanding of the building blocks of a Windows Sockets application. Following the walkthrough, we will discuss the use of Windows Sockets by the WormHole sample application.

The Basic Structures

Although the Windows Sockets specification contains about a dozen different structures, application developers will quickly become familiar with a few that are required by nearly all Windows Sockets applications.

```
struct sockaddr {

        u_short sa_family;

        char    sa_data[14];

};

struct sockaddr_in {

        short   sin_family;

        u_short sin_port;

        struct  in_addr sin_addr;

        char    sin_zero[8];

};
```

The *sockaddr* structure is used by Windows Sockets to specify a local or remote endpoint address to which to connect a socket. An endpoint address simply contains the appropriate information to send data between two sockets on different systems. As the contents of endpoint addresses differ between network protocol families, the *sockaddr* structure was designed to accommodate endpoint addresses of variable size, satisfying requirements of many common network protocol families. The first field of a *sockaddr* contains the family number identifying the format of the remaining part of the address.

In the Internet address family, the *sockaddr_in* structure is used to store the endpoint address information and is cast to type *sockaddr* for the functions which require it. Other address families must define their own *sockaddr_* structures as appropriate for their needs. For TCP/IP, the *sockaddr_in* structure breaks the endpoint address into its two components: port ID (sin_port) and IP address (sin_addr), and pads the remaining eight bytes of the endpoint address with a character string (sin_zero). The port and IP address values are always specified in network byte order. The value for *sin_family* under TCP/IP is always *AF_INET* (address family Internet).

```
struct  hostent {

        char FAR *              h_name;

        char FAR * FAR *        h_aliases;

        short                       h_addrtype;

        short                       h_length;

        char FAR * FAR *        h_addr_list;

};
```

The *hostent* structure is generally used by the Windows Sockets database routines to return *host*, or system, information about a specified system on the network. The host structure contains the primary name for a system including optional aliases for the primary name. Additionally, it contains a list of address(es) for the specified system. This information is generally sought for the remote system to which

an application is connecting using the Windows Sockets database routines described later.

```
struct  protoent {

        char FAR *              p_name;

        char FAR * FAR *        p_aliases;

        short                   p_proto;

};


struct  servent {

        char FAR *              s_name;

        char FAR * FAR *        s_aliases;

        short               s_port;

        char FAR *              s_proto;

};
```

The *protoent* and *servent* structures are also filled by the Windows Sockets database routines. These structures contain information about a particular protocol (TCP, or UDP,) or service (finger or telnet, for example) respectively. Along with the primary name and an array of aliases for the protocol or service, these structures also contain their corresponding 16-bit IDs, necessary to build a valid TCP/IP endpoint address.

```
typedef struct WSAData {

        WORD                    wVersion;

        WORD                    wHighVersion;

        char                    szDescription[WSADESCRIPTION_LEN+1];

        char                    szSystemStatus[WSASYS_STATUS_LEN+1];

        unsigned short          iMaxSockets;

        unsigned short          iMaxUdpDg;

        char FAR      *             lpVendorInfo;

} WSADATA;
```

Finally, the *WSAData* structure is filled in by a Windows Sockets DLL when an application calls the *WSAStartup()* API. Along with Windows Sockets version information, the structure also contains vendor-specific information, such as the maximum number of sockets available and the maximum datagram size. The *szDescription* and *szSystemStatus* members can be used by an implementation to identify itself and the current status of the DLL. For example, an implementation may return the text "Joe's ShareWare Windows Sockets implementation v1.2. 10/22/92" in *szDescription.* The specification of the *lpVendorInfo* member is completely up to an implementor and is not defined in the Windows Sockets specification.

Setting Up, and Cleaning Up Your Windows Sockets Application

As mentioned earlier, Windows Sockets offers some extensions to the Berkeley sockets paridigm to allow your application to be more "friendly" in the Windows Environment. All such functions are preceded by the characters "WSA", which is short for "Windows Sockets API." Although the use of WSA functions is strongly advised, there are two WSA functions that your application can't avoid: *WSAStartup()* and *WSACleanup()*.

WSAStartup() "attaches" your application to Windows Sockets and causes the Windows Sockets DLL to initialize any structures that it might need for operation. Additionally, *WSAStartup()* performs version negotiation and forces an internal Windows Sockets reference count to be incremented. This reference count allows Windows Sockets to maintain the number of applications on the local system requiring Windows Sockets services and structures. The version negotiation allows an application to determine whether or not the underlying Windows Sockets implementation is able to support the same version of the Windows Sockets specification that the application is written to. A Windows Sockets implementation may or may not support multiple versions of the specification. Other Windows Sockets-specific information may also be filled in such as the vendor of the implementation, the maximum datagram size supported, maximum number of sockets which an application can open, and more.

The following startup code is authored to run only under version 1.1 (the most current version of the Windows Sockets specification) or later, and requires that at least six sockets be available to the calling application.

```
#define  WS_VERSION_REQD    0x0101

#define  WS_VERSION_MAJOR   HIBYTE(WS_VERSION_REQD)

#define  WS_VERSION_MINOR   LOBYTE(WS_VERSION_REQD)

#define  MIN_SOCKETS_REQD   6


WSADATA  wsaData;

char  buf[MAX_BUF_LEN];

int   error;



.

.

.

error=WSAStartup(WS_VERSION_REQUIRED,&wsaData);

if (error !=0 ) {

   /* Report that Windows Sockets did not respond to the WSAStartup() call
*/



   sprintf(buf,"winsock.dll not responding.");

   MessageBox (hWnd, buf. "Windows Sockets Error",MB_OK);
```

```
   shutdown_app();
}

if (( LOBYTE (wsaData.wVersion) < WS_VERSION_MAJOR) ||
   ( LOBYTE (wsaData.wVersion) == WS_VERSION_MAJOR &&
     HIBYTE (wsaData.wVersion) < WS_VERSION_MINOR)) {

   /* Report that the application requires Windows Sockets version
WS_VERSION_REQD */
   /* compliance and that the winsock.dll on the system does not support
it.       */

   sprintf(buf,"Windows Sockets version %d.%d not supported by
winsock.dll",
     LOBYTE (wsaData.wVersion), HIBYTE (wsaData.wVersion));
   MessageBox (hWnd, buf. "Windows Sockets Error",MB_OK);
   shutdown_app();
}

if (wsaData.iMaxSockets < MIN_SOCKETS_REQUIRED ) {

   /* Report that winsock.dll was unable to support the minimum number of
*/
   /* sockets (MIN_SOCKETS_REQD) for the application
*/

   sprintf(buf,"This application requires a minimum of %d supported
sockets.",
     MIN_SOCKETS_REQUIRED);
   MessageBox (hWnd, buf. "Windows Sockets Error",MB_OK);
   shutdown_app();
}
.
```

A Windows Sockets application generally calls*WSACleanup()* during its own cleanup, decrementing the internal reference count and letting Windows Sockets know that it's no longer needed by the calling application. Whatever cleanup this function forces is implementation-specific and shielded from the application. The application author should, however, check for any possible error conditions from *WSACleanup()* and report them before exiting, as this information might indicate a network layer problem in the system.

Error Handling

In order to provide a consistent mechanism for reporting errors and to ensure safety of Windows Sockets applications in multithreaded versions of Windows (like Windows NT), the *WSAGetLastError()* API was introduced as a means to get the code for the last network error on a particular thread. Under Windows 3.x, thread safety is not an issue, although *WSAGetLastError()* is still the appropriate way to check for extended error codes. Many functions in the Windows Sockets API set return an error code in the event that there was a problem, and rely on the application to call *WSAGetLastError()* to get more detailed information on the failure. The following code illustrates how an application might report an error to a user:

```
LPHOSTENT   host_info;

char     user_buf[MAX_BUF], appl_buf[MAX_BUF];

.

.

.

/* Attempt to resolve hostname specified by user_buf,return meaningful */

/* message to the user in the event of an error. */


host_info=gethostbyname(user_buf);

if(host_info==NULL){

   sprintf(buf,"Windows Sockets error %d: Hostname: %s couldn't be
resolved.",

     WSAGetLastError(),user_buf);

   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);}

.

.

.
```

In addition to the *WSAGetLastError()* API, an application may choose to use the *WSASetLastError()* API to set a network error condition which will be returned by a subsequent *WSAGetLastError()* call. Obviously, anyWindows Sockets calls made between a *WSASetLastError()* and *WSAGetLastError()* pair will override the code set by the *WSASetLastError()* call.

Database Routines

The TCP/IP protocol relies on the binary representations for addresses and various other identifiers. However, end users and programmers prefer to use easy-to-remember names (such as ftp, rhino.microsoft.com). It is therefore necessary to provide a common method to resolve both services and hostnames into their respective binary equivalents. To solve this, the Windows Sockets specification offers a set of APIs known as the *database routines.*

The database routines fall into three categories

host resolution	Learning the IP address for a host based on system, or host name
protocol resolution	Learning the protocol ID of a specific member of a protocol family (TCP, for example)
service resolution	Learning the port ID of a service based on a service name/protocol pair

All the database routines return information in structures defined in the previous section.

Applications use the *gethostbyname()* and *gethostbyaddr()* functions to learn about the names and IP address(es) of a particular system, knowing only the name or the address of the system. Both calls return a pointer to a *hostent* structure as defined in the previous section. The *gethostbyname()* call simply accepts a pointer to a null-terminated string representing the name of the system to resolve. The *gethostbyaddr()* instead accepts three parameters: a pointer to the address (in network byte order), the length of the address, and the type of address.

Generally the hostname or IP address is offered to the application by the user to specify a remote system to connect to, and the IP address is resolved by Windows Sockets by either parsing a local *hosts* file, or querying a DNS (domain name system) server. The details of the resolution however, are specific to the implementation, abstracted from the application by these APIs.

The *getservbyname()* and *getservbyport()* functions return information about well-known Windows Sockets services, or applications. Each of these system calls return a pointer to a *servent* structure, as defined in the previous section. Typically an application will use these calls to determine the port ID for a well-known service (such as FTP) to create an endpoint address.

The following code fragment demonstrates the use of the *getservbyname()* function to fill in the *sockaddr_in* structure which will be used to connect a socket to a well-known port (the FTP protocol port over TCP):

```
char      buf [MAX_BUF_LEN];

struct sockaddr_in          srv_addr;

LPSERVENT       srv_info;

LPHOSTENT       host_info;

SOCKET      s;

.

.

.

/* Get FTP service port information */
```

```
srv_info=getservbyname("ftp","tcp");


if (srv_info== NULL) {

   /* Couldn't find an entry for "ftp" over "tcp" */


   sprintf(buf,"Windows Sockets error %d: Couldn't resolve FTP service
port.",

     WSAGetLastError());

   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);

   shutdown_app();

}


/* Set up socket */


srv_addr.sin_family = AF_INET;

srv_addr.sin_addr.s_addr = INADDR_ANY;

srv_addr.sin_port=srv_info->s_port;
```

The example uses *getservbyname()* to resolve the port number of the FTP service over TCP. The port ID is used to construct the *sockaddr_in* structure (endpoint address) for future use by the application. As we mentioned before, the address family for TCP/IP is always assigned as AF_INET. We use the *INADDR_ANY* macro to specify any local IP interface to accept incoming connections (more on this later).

To round out the database routines, *getprotobyname()* and *getprotobynumber()* fill in a *protoent* structure, sometimes used by applications to create a socket over a particular protocol (e.g., UDP or TCP). More often, however, an application will use the SOCK_DGRAM and SOCK_STREAM macros to create either datagram or stream sockets.

Data Manipulation Routines

There are several routines that convert values between network byte order and host (or local system) byte order. Windows Sockets offers byte-ordering routines for 16- and 32-bit values from both host byte order and network byte order. The *htons()* function takes a 16-bit value (a short), and converts it from host byte order to network byte order – hence the name htons (host to network short). The other byte-ordering functions available are *htonl()*, *ntohs()*, and *ntohl()*.

There are two other useful routines offered by Windows Sockets which convert IP addresses between strings and network byte-ordered 32-bit values. These functions are *inet_ntoa()* and *inet_addr()*. These routines are useful to convert the IP address of an endpoint user input. In the following example, a TCP-based server application uses the *inet_ntoa()* function to log incoming connection attempts:

```
SOCKET      cli_sock, srv_sock;

LPSOCKADDR_IN         cli_addr;

char      *cli_ip,    buf[MAX_BUF];

int       len;

.

/* Accept incoming connection, create new local socket cli_sock */


cli_sock=accept(srv_sock,(LPSOCKADDR)&cli_addr,&len);


if (cli_sock==INVALID_SOCKET){

   return(ERROR);

}


/* Convert endpoint IP address from network byte order to ASCII */


cli_ip=inet_ntoa(cli_addr.sin_addr);

sprintf(buf,"Incoming connection request from: %s.\n",cli_ip);

log_event(buf);
```

Setting Up Client and Server Sockets

Most Windows Sockets applications are asymmetrical; that is, there are generally two components to the network application – a client and a server. Frequently, these components are isolated into separate programs. Sometimes, these components are integrated into a single application (such as our sample application). When both the client and server components of a networking application are integrated, the application is generally referred to as a "peer" application. Both the client and the server components go through different procedures to ready themselves for networking by making a number of Windows Sockets API calls. The following state diagrams illustrate the state transitions for setting up client- and server-side socket applications:

{ewc msdncd, EWGraphic, bsd23459 4 /a "SDKD.BMP"}

Setting up a server-side stream-based application

{ewc msdncd, EWGraphic, bsd23459 5 /a "SDKD.BMP"}

Setting up a client-side stream socket-based application

The *socket()* call creates an endpoint for both client- and server-side application communication. When calling *socket()*, the application specifies the protocol family and either the socket type (stream or datagram), or the specific protocol which it expects to use (for example, TCP). Both the client- and server-side of a network application use the *socket()* call to define their respective endpoints. *socket()* returns a *socket descriptor*, an integer which uniquely identifies the socket created within Windows Sockets.

Server-side connection setup

Once the socket is created, the server-side associates the freshly created socket descriptor and a *local endpoint address* via the *bind()* API. The local endpoint address is comprised of two pieces of data, the IP address and the port ID for the socket. The local IP address is used to determine which interfaces the server application will accept connection requests on; the port ID identifies the TCP or UDP port on which connections will be accepted. It is for these two values that the network byte-ordering routines (*htonl(), htons(),* etc..) were created. These values must always be represented in network byte order.

Alternatively, an application may substitute the value INADDR_ANY in place of a valid local IP address, and the system will accept incoming requests on any local interface, and will send requests on the "most appropriate" local interface. In fact, most server applications do exactly this. To associate a socket with any valid system port, provide a value of 0 for the *.sin_port* member of the *sockaddr_in* structure. This will select an unused system port between 1025 and 5000. As mentioned before, most server applications listen on a specified port, and client applications use this mechanism to obtain an unused local port. Once an application uses this mechanism to obtain a valid local port, it may call *getsockname()* to determine the port the system selected.

The *listen()* API sets up a connection queue. It accepts only two parameters, the socket descriptor and the queue length. The queue length identifies the number of outstanding connection requests that will be allowed to queue up on a particular port/address pair, before denying service to incoming connections.

The *accept()* API completes a stream-based server-side connection by accepting an incoming connection request, assigning a new socket to the connection, and returning the original socket to the *listen*ing state. The new socket is returned to the application, and the server can begin interacting with the client over the network.

Client-side connection setup

From the client's perspective, the application also creates a socket using the *socket()* call. The *bind()* command is used to bind the socket to a locally specified endpoint address which the server will use to transmit data back to the client. Once a local endpoint association is made, the *connect()* API establishes a connection with a remote endpoint. This routine initiates the network connection between the two

systems. Once the connection is made, the client can begin interaction with the server on the network.

Although the client may choose to call **bind()**, it is not necessary to do so. Calling **connect()** with an unbound socket will simply force the system to choose an IP interface and unique port ID and mark the socket as bound. Most client-side applications neglect the **bind()** call as there are rarely specific requirements for a particular local interface/port ID pair.

The following code fragments create and connect a pair of stream-based sockets using the API flow outlined above:

Server-side (connection-oriented)

```
#define     SERVICE_PORT          5001


SOCKET      srv_sock,       cli_sock;

struct sockaddr_in          srv_addr,         cli_addr;

char      buf[MAX_BUF_LEN];


/* Create the server-side socket */


srv_sock=socket(AF_INET,SOCK_STREAM,0);

if (srv_sock==INVALID_SOCKET){

   sprintf(buf,"Windows Sockets error %d: Couldn't create socket.",

     WSAGetLastError());

   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);

   shutdown_app();

}


srv_addr.sin_family=AF_INET;

srv_addr.sin_addr.s_addr=INADDR_ANY;

srv_addr.sin_port=SERVICE_PORT;         /* specific port for server to
listen on */


/* Bind socket to the appropriate port and interface (INADDR_ANY) */


if (bind(srv_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR){

   sprintf(buf,"Windows Sockets error %d: Couldn't bind socket.",
```

```
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}

/* Listen for incoming connections */

if (listen(srv_sock,1)==SOCKET_ERROR){

   sprintf(buf,"Windows Sockets error %d: Couldn't set up listen on
socket.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}

/* Accept and service incoming connection requests indefinitely */

for ( ; ; ) {
   cli_sock=accept(srv_sock,(LPSOCKADDR)&cli_addr,&addr_len);
   if (cli_sock==INVALID SOCKET){

    sprintf(buf,"Windows Sockets error %d: Couldn't accept incoming \
        connection on socket.",WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
   }
   .
   /* Client-server network interaction takes place here */
   .
   closesocket(cli_sock);
```

```
}
```

Client-side (connection-oriented)

```
/* Static IP address for remote server for example. In reality, this would
be
   specified as a hostname or IP address by the user */

#define  SERVER       "131.107.1.121"

struct   sockaddr_in  srv_addr,cli_addr;
LPSERVENT       srv_info;
LPHOSTENT       host_info;
SOCKET      cli_sock;
.
/* Set up client socket */

cli_sock=socket(PF_INET,SOCK_STREAM,0);

if (cli_sock==INVALID_SOCKET){

   sprintf(buf,"Windows Sockets error %d: Couldn't create socket.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}

cli_addr.sin_family=AF_INET;
cli_addr.sin_addr.s_addr=INADDR_ANY;
cli_addr.sin_port=0;                    /* no specific port req'd */

/* Bind client socket to any local interface and port */
```

```
if (bind(cli_sock,(LPSOCKADDR)&cli_addr,sizeof(cli_addr))==SOCKET_ERROR){

   sprintf(buf,"Windows Sockets error %d: Couldn't bind socket.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}

/* Get the remote port ID to connect to for FTP service */

srv_info=getservbyname("ftp","tcp");

if (srv_info== NULL) {

   sprintf(buf,"Windows Sockets error %d: Couldn't resolve FTP service
port.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}

srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = inet_addr(SERVER);
srv_addr.sin_port=srv_info->s_port;

/* Connect to FTP server at address SERVER */

if (connect(cli_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR)
{

   sprintf(buf,"Windows Sockets error %d: Couldn't connect socket.",
     WSAGetLastError());
```

```
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);

    shutdown_app();

}

/* Client-server network interaction takes place here */
```

Although the above fragment makes use of the **bind()** API, it would be just as effective to skip over this call as there are no specific local port ID requirements for this client. The only advantage that **bind()** offers is the accessibility of the port which the system chose via the **.sin_port** member of the **cli_addr** structure which will be set upon success of the **bind()** call.

For connectionless, or "datagram" sockets, Windows Sockets usage is a little simpler. Since the communication in datagram sockets is connectionless, it is not necessary to use the APIs necessary for creating a connection, namely: *connect()*, *listen()*, and *accept().* The flow of Windows Sockets APIs that a typical connectionless client-server pair will generally traverse follows:

{ewc msdncd, EWGraphic, bsd23459 6 /a "SDKD.BMP"}

Setting up a server-side datagram-based application

{ewc msdncd, EWGraphic, bsd23459 7 /a "SDKD.BMP"}

Setting up a client-side stream-based application

As pictured above, a client may choose to *connect()* the datagram socket for convenience of multiple sends to the remote endpoint. Connecting a datagram socket will cause all sends to go to the connected address, and any datagrams received from a remote address different than the connected address are discarded by the system. Generally, connectionless clients use the *sendto()* API to transmit application data. The *sendto()* call requires that the destination's endpoint address be specified with every call to the API. By *connect*ing a datagram socket, a client sending a large amount of data to the same destination can simply use the *send()* API to transmit, without having to specify a remote endpoint with every call, and the client need not concern itself with the possibility of receiving unwanted data from other hosts. Depending on the type of application you are developing, and the Windows Sockets implementation below your application, *connect*ing datagram sockets may improve performance of your application.

Some sample code illustrates how the TFTP protocol (a connectionless protocol for file transfer) client and server might be implemented over Windows Sockets:

Server-side (connectionless)

```
SOCKET      srv_sock;

struct sockaddr_in          srv_addr;


.

.

/* Create server socket for connectionless service */


srv_sock=socket(PF_INET,SOCK_DGRAM,0);
```

```
if (srv_sock==INVALID_SOCKET){


   sprintf(buf,"Windows Sockets error %d: Couldn't create socket.",

     WSAGetLastError());

   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);

   shutdown_app();

}


/* Resolve TFTP service port to listen on */


srv_info=getservbyname("tftpd","udp");


if (srv_info== NULL) {


   sprintf(buf,"Windows Sockets error %d: Couldn't resolve TFTPd service
port.",

     WSAGetLastError());

   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);

   shutdown_app();

}


srv_addr.sin_family = AF_INET;

srv_addr.sin_addr.s_addr = INADDR_ANY;   /* Allow the server to accept
connections                                          /* over any
interface */

srv_addr.sin_port=srv_info->s_port;


/* Bind remote server's address and port */


if (bind(srv_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR){


   sprintf(buf,"Windows Sockets error %d: Couldn't bind socket.",
```

```
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}

/* Client-server network interaction takes place here */
```

Client-side (connectionless)

```
/* Static IP address for remote server for example. In reality, this would
be
   specified as a hostname or IP address by the user */

#define  SERVER       "131.107.1.121"

struct sockaddr_in          srv_addr,cli_addr;
LPSERVENT       srv_info;
LPHOSTENT       host_info;
SOCKET      cli_sock;

.
.
/* Create client-side datagram socket */

cli_sock=socket(PF_INET,SOCK_DGRAM);
if (cli_sock==INVALID_SOCKET){

   sprintf(buf,"Windows Sockets error %d: Couldn't create socket.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}
```

```
cli_addr.sin_family=AF_INET;
cli_addr.sin_addr.s_addr=INADDR_ANY;
cli_addr.sin_port=0;                    /* no specific local port req'd */

/* Bind local socket */
if (bind(cli_sock,(LPSOCKADDR)&cli_addr,sizeof(cli_addr))==SOCKET_ERROR){

   sprintf(buf,"Windows Sockets error %d: Couldn't bind socket.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}

/* Resolve port information for TFTP service */

srv_info=getservbyname("tftp","udp");
if (srv_info== NULL) {

   sprintf(buf,"Windows Sockets error %d: Couldn't resolve TFTP service
port.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_app();
}

srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = inet_addr(SERVER);
srv_addr.sin_port=srv_info->s_port;

if (connect(cli_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR)
{
```

```
    sprintf(buf,"Windows Sockets error %d: Couldn't connect socket.",
      WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
}
/* Client-server network interaction takes place here */
```

Since a connectionless client (such as TFTP) will undoubtedly be doing successive transmissions with the server (especially during long transfers), we have chosen to connect the socket, allowing the use of the **send()** and **recv()** APIs rather than **sendto()** and **recvfrom()**. The use of **connect()** with datagram sockets is purely optional.

Sending and Receiving Data (stream sockets)

Once an application successfully establishes a socket connection, it is ready to start transferring data over the connection. With stream (TCP) sockets, data transfer is said to be "reliable," meaning that the application may assume that the underlying transport will ensure that the data gets to the remote host without duplication or corruption. When a connection is established on a stream socket, a the TCP transport creates a "virtual circuit" between the two machines. This circuit remains open until both applications decide that they are done sending data on the circuit (typically a "graceful" close), or until a network error occurs which causes the circuit to be terminated abnormally.

An application sends data using the *send()* API. This API takes a socket descriptor, a pointer to a buffer to send, the length of the buffer, and an integer which specifies flags which can modify the behavior of *send().* To receive data, an application uses the *recv()* API, which takes a pointer to a buffer to fill with data, the length of the specified buffer, and a flags integer. *recv()* returns the number of bytes actually received, and *send()* returns the number of bytes actually sent. Note that applications should always check the return codes of *send()*and *recv()* for the number of bytes actually transferred, since it may be different from the number requested. Because of the stream-oriented nature of TCP sockets, there isn't necessarily a one-to-one correspondence between *send()* and *recv()* calls. For example, a client application may perform ten calls to *send()*, each for 100 bytes. The system may combine or "coalesce" these sends into a single network packet, so that if the server application did a *recv()* with a buffer of 1000 bytes, it would get all the data at once.

Therefore, an application must not make any assumptions about how data will arrive. A server which expects to receive 1000 bytes should call *recv()*in a loop until it has received all of the data:

```
SOCKET    s;

int       bytes_received;

char      buffer[1000];

char      *buffer_ptr;

int       buffer_length;


.

.

buffer_ptr = buffer;

buffer_length = sizeof(buffer);


/* Receive all outstanding data on socket s */


do {

    bytes_received = recv(s, buffer_ptr, buffer_length, 0);

    if (bytes_received == SOCKET_ERROR) {

    sprintf(buf,"Windows Sockets error %d: Error while receiving data.",
```

```
        WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
   }
    buffer_ptr += bytes_received;
    buffer_length -= bytes_received;
} while (buffer_length > 0);
```

Likewise, an application which wants to send 1000 bytes should *send()* in a loop until all of the data has been sent:

```
SOCKET    s;
int       bytes_sent;
char      buffer[1000];
char      *buffer_ptr;
int       buffer_length;

buffer_ptr = buffer;
buffer_length = sizeof(buffer);

/* Enter send loop until all data in buffer is sent */

do {
    bytes_sent = send(s, buffer_ptr, buffer_length, 0);
    if (bytes_sent == SOCKET_ERROR) {
    sprintf(buf,"Windows Sockets error %d: Error while sending data.",
        WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_app();
   }
    buffer_ptr += bytes_sent;
    buffer_length -= bytes_sent;
} while (buffer_length > 0);
```

Sending and Receiving Data (datagram sockets)

Data transmission on datagram sockets is significantly different from stream sockets. The most important difference is that transmission is not reliable on datagram sockets. This means that if an application attempts to send data to another application, the system does nothing to guarantee that the data will actually be delivered to the remote application. Reliability will tend to be good on LANs (local-area networks, such as a network connecting several computers within a single building), but can be very poor on WANs (wide-area networks, such as the Internet which connects hundreds of thousands of computers worldwide).

The next important difference in datagram sockets is that they are connectionless. This means that there is no default remote address assigned to them. For this reason, an application which wants to send data must specify the address to which the data is destined with the *sendto()* API. In addition, an application typically wants to know where data came from, so it receives data with the *recvfrom()* API, which returns the address of the sender of the data.

The final significant difference in data transmission for datagram sockets is that it is "message oriented." This means that there is a one-to-one correspondence between *sendto()* and *recvfrom()* calls, and that the system does not coalesce data when sending it. For example, if an application makes 10 calls to *sendto()* with a buffer of two bytes, the remote application will need to perform 10 calls to *recvfrom()* with a buffer of at least two bytes to receive all the data.

Below is a code fragment from a datagram sockets application which echos datagrams back to the sender.

```
SOCKET      s;

SOCKADDR_IN           remoteAddr;

int       remoteAddrLength = sizeof(remoteAddr);

BYTE      buffer[1024];

int       bytesReceived;


for ( ; ; ) {

   /* Receive a datagram on socket s */


   bytesReceived = recvfrom( s, buffer, sizeof(buffer),0,

                        (PSOCKADDR)&remoteAddr, &remoteAddrLength );


   /* Echo back to the server as long as bytes were received */


   if ( bytesReceived != SOCKET_ERROR ||

        sendto( s, buffer, bytesRecieved, 0,
```

```
                (PSOCKADDR)&remoteAddr, remoteAddrLength ) ==
          SOCKET_ERROR ){
      sprintf(buf,"Windows Sockets error %d: Error while sending data.",
         WSAGetLastError());
      MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
      shutdown_appl();
     }
  }
```

As mentioned earlier, it is possible to *connect()* a datagram socket. Once a datagram socket is connected, all data is sent and received from the remote address to which the datagram is connected, so it is possible to use the *send()* and *recv()* APIs on connected datagram sockets.

Terminating a Connection

An application has several options for terminating a connection. The simplest is to call the *closesocket()* API, which takes only a socket descriptor as input. This API frees resources associated with a socket and initiates the graceful close sequence. This sequence is completed when the remote application also closes its socket.

If an application determines that it is done sending data, but may want to receive more data, it can call the *shutdown()* API. This API notifies the remote end that the local application won't be sending any more data, but may continue to receive data.

Lastly, an application may cause an "abortive" or "hard" close on a connection with the SO_LINGER socket option in conjunction with *closesocket()*. Setting the linger timeout to 0 causes the circuit to be terminated immediately, regardless of whether the remote end has completed its data transfer, and any unreceived or unsent data is dropped. Therefore, this option should be used with caution and only if the results are understood and intended. The following code fragment demonstrates how to perform a hard close on a connection:

```
LINGER    lingerInfo;

INT       err;

SOCKET    s;


/* First set the linger timeout on the socket to 0.  This will */

/* cause the connection to be reset. */


lingerInfo.l_onoff = 1;

lingerInfo.l_linger = 0;


setsockopt( s, SOL_SOCKET, SO_LINGER, (char
*)&lingerInfo,sizeof(lingerInfo) );

closesocket(s);
```

How can an application know that the remote end has terminated the connection? The answer depends on whether the remote end terminated the connection gracefully or abortively. If the termination was abortive, then *send()* and *recv()* calls will fail with the error WSAECONNRESET. This indicates to an application that data may have been lost, and the error condition should be reported to the user.

If the termination was graceful, then any *recv()* calls made after all data has been received will return the value zero as the number of bytes received. This indicates that the remote end has gracefully terminated its end of the connection and the local end may close the socket without fear of data loss. The following code fragment illustrates how an application initiates a graceful close and then waits for the remote end to close gracefully before closing the socket.

```
SOCKET    s;

INT       err;

BYTE      buffer[1024];
```

```
err = shutdown( s , 1 );
if ( err == SOCKET_ERROR ) {
   sprintf(buf,"Windows Sockets error %d: Error during shutdown.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_appl();
}

/* Receive the rest of the pending data */

while ( (err = recv( s, buffer, sizeof(buffer), 0 ) != 0 ) {

   if ( err == SOCKET_ERROR ) {
    sprintf(buf,"Windows Sockets error %d: Error while receiving data.",
       WSAGetLastError());
    MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
    shutdown_appl();
   }
   .
   .
    /* do something with the data we received. */
   .
   .

}

/* The other side has also terminated.  we can safely close now */

closesocket( s );
```

Note that in this example the application keeps calling *recv()* until it returns zero bytes received. If an

application closes a socket and there is data available to be received, or data later arrives, then the system will abort the connection and throw out the data, since there is nobody to give the data to. Well-behaved applications should ensure that they receive all data before closing a socket.

The Windows Sockets Extensions

Asynchronous Windows Sockets Calls

By default, an application's socket calls will block until the requested operation can be completed. For example, if an application wishes to receive data from another application, its call to the *recv()* API will not complete until the other application has sent data which can be returned to the calling application.

This model is sufficient for simple applications, but more sophisticated applications may not wish to block for an arbitrarily long period for a network event. In fact, in Windows 3.1, blocking operations are considered poor programming practice because applications are expected to call *PeekMessage()* or *GetMessage()* regularly in order to allow other applications to run and to receive user input.

To support the sophisticated applications which fit better within the Windows programming paradigm, Windows Sockets supports the concept of "nonblocking sockets." If an application sets a socket to nonblocking, then any operation which may block for an extended period will fail with the error code WSAEWOULDBLOCK. This error indicates to the application that the system was unable to perform the requested operation immediately.

How, then, does an application know when it can successfully perform certain operations? Polling would be one (poor) solution. The optimal mechanism is to use the asynchronous notification mechanism provided by the *WSAAsyncSelect()* API. This routine allows an application to notify a Windows Sockets implementation of certain events which are of interest, and to receive a Windows message when the events occur. For example, an application may indicate interest in data arrival with the FD_READ message, and when data arrives the Windows Sockets DLL posts a message to the application's window handle. The application receives this message in a *GetMessage()* or *PeekMessage()* call and can then perform the corresponding operation.

The following code fragment demonstrates how an application opens and connects a TCP socket and indicates that it is interested in being notified when one of three network events occurs:

- data arrives on the socket
- it is possible to send data on the socket
- the remote end has closed the socket

```
/* Static IP address for remote server for example. In reality, this would
be

   specified as a hostname or IP address by the user */


#define  SERVER             "131.107.1.121"


#define  SOCKET_MESSAGE     WM_USER+1

#define  SERVER_PORT        4000


struct sockaddr_in          srv_addr;

SOCKET      cli_sock;
```

```
.
.
.
/* Create client-side socket */

cli_sock=socket(PF_INET,SOCK_STREAM,0);

if (cli_sock==INVALID_SOCKET){
   sprintf(buf, "Windows Sockets error %d: couldn't open socket.",
     WSAGetLastError());
   MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_appl();
}

srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = inet_addr(SERVER);
srv_addr.sin_port=SERVER_PORT;

/* Connect to server */

if (connect(cli_sock,(LPSOCKADDR)&srv_addr,sizeof(srv_addr))==SOCKET_ERROR)
{

   sprintf(buf,"Windows Sockets error %d: Couldn't connect socket.",
     WSAGetLastError());
   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);
   shutdown_appl();
}

/* Set up async select on FD_READ, FD_WRITE, and FD_CLOSE events */
```

```
err = WSAAsyncSelect(cli_sock, hWnd, SOCKET_MESSAGE, FD_READ|FD_WRITE|
FD_CLOSE);

if (err == SOCKET_ERROR) {

   sprintf(buf,"Windows Sockets error %d: WSAAsyncSelect failure.",

     WSAGetLastError());

   MessageBox (hWnd,buf,"Windows Sockets Error",MB_OK);

   shutdown_appl();

}
```

When one of the specified network events occurs, the specified window handle *hWnd* receives a message containing a *wMsg* of SOCKET_MESSAGE. The *wParam* field of the message will have the socket handle, and *lParam* contains two pieces of information: the low word contains the event that occurred (FD_READ, FD_WRITE, or FD_CLOSE) and the high word contains an error code, or 0 if there was no error. Code similar to the following fragment, which would belong in an application's main window procedure, may be used to interpret a message from *WSAAsyncSelect()*:

```
long FAR PASCAL _export WndProc(HWND hWnd, UINT message, UINT wParam, LONG
lParam)

{

   INT err;


   switch (message) {


   case ...:

    /* handle Windows messages */

   .

   .

   .

   case SOCKET_MESSAGE:


    /* A network event has occurred on our socket.  Determine which network


    switch (WSAGETSELECTEVENT(lParam)) {

    case FD_READ:

      /* Data arrived.  Receive it. */

      err = recv(cli_sock, Buffer, BufferLength, 0);
```

```
   if (WSAGetLastError() == WSAEWOULDBLOCK) {
      /* We have already received the data. */
      break;
   }
   if (err == SOCKET_ERROR) {
      sprintf(buf, "Windows Sockets error %d: receive error.",
      MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);
      shutdown_appl();
   }
   .
   .
   /* Do something useful with the data. */
   .
   .
   break;

case FD_WRITE:
   /* We can send data. */

   err = send(cli_sock, Buffer, BufferLength, 0);
   if (err == SOCKET_ERROR) {
      if (WSAGetLastError() == WSAEWOULDBLOCK) {
       /* Send buffers overflowed. */
       break;
      }
      sprintf(buf, "Windows Sockets error %d: send failed.",
        SAGetLastError());
      MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);
      shutdown_appl();
   }
   break;
```

```
case FD_CLOSE:
  /* The remote closed the socket. */


  closesocket(cli_sock);
  break;
}


break;
}
```

.

An important part of the behavior of *WSAAsyncSelect()* is that after the Windows Sockets DLL has posted a message for a particular network event, the DLL refrains from posting another message for that event until the application has called the "reenabling function" for that event. For example, once an FD_READ has been posted, no more FD_READ messages will be posted until the application calls *recv()*. This prevents an application's message queue from being overflowed with messages for a single network event.

Using *WSAAsyncSelect()* can simplify and improve organization in Windows applications by allowing them to be fully event-driven. Such an application responds to network events in much the same way it responds to user events such as a mouse click. In addition, applications which make use of WSAAsyncSelect() are better behaved Windows applications since they must frequently call *PeekMessae()* or *GetMessage()* in order to receive the network messages.

Asynchronous Database Routines

Just as simple network operations like *recv()* can block for extended periods of time, so can the database routines like *gethostbyname()*, especially is they go through DNS which can require considerable time due to the network activity involved. In order to allow well-behaved Windows Applications to use the database routines, the Windows Sockets API supplies asynchronous versions of the database routines. Their use is similar to the synchronous database routines with two important exceptions:

- Completion of the operation is indicated via a Windows message, as with *WSAAsyncSelect().*
- The application is required to include a buffer which is filled in by the Windows Sockets DLL with the information requested by the application. This is in contrast to the synchronous database routines which return a pointer to space owned by the Windows Sockets DLL.

In order to make an asynchronous database call, an application passes in information about the call, a window handle to receive the completion message, a message code, and a buffer for the output information. The Windows Sockets API defines a constant, MAXGETHOSTSTRUCT, which is the maximum size of a hostent structure. An application should use a buffer of this size to pass to the asynchronous database routines in order to be guaranteed that all the output information will fit in its buffer. A call to *WSAAsyncGetHostByName()* may be as follows:

```
#define GETHOST_MESSAGE          WM_USER+2


BYTE        HostBuffer[MAXGETHOSTSTUCT];

HANDLE TaskHandle;


/* Resolve hostname "hostname" asynchronously */


TaskHandle = WSAAsyncGetHostByName(hWnd, GETHOST_MESSAGE, "hostname",
HostBuffer,                                        MAXGETHOSTSTUCT);


if (TaskHandle == SOCKET_ERROR) {


   sprintf(buf, "Windows Sockets error %d: Hostname couldn't be resolved.",
         WSAGetLastError());

   MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);

   shutdown_appl();

}
```

WSAAsyncGetHostByName() and the other asynchronous database routines return a "task handle" which uniquely refers to the operation in progress. This task handle allows an application which may have multiple outstanding asynchronous database routines to associate completion messages with the request, since the task handle is returned in the completion message as *wParam.*

To receive and process the completion message, an application will do the following in its window

procedure:

```
long FAR PASCAL _export WndProc(HWND hWnd, UINT message, UINT wParam, LONG
lParam)

{

   INT err;


   switch (message) {

   case ...:

    /* handle Windows messages */

   .

   .

   .


   case GETHOST_MESSAGE:


    /* An asynchronous database routine completed. */

    if (WSAGETASYNCERROR(lParam) != 0) {

      sprintf(buf, "Windows Sockets error %d: Hostname couldn't be

      MessageBox(hWnd,buf,"Windows Sockets Error",MB_OK);

      shutdown_appl();

    }

    .

    .

    /* HostBuffer now contains a host buffer, use info from it. */

    .

    .

}
```

Windows Sockets on Windows NT

Although the initial focus of Windows Sockets was for Windows 3.1 16-bit applications, Windows NT supports Windows Sockets as well. To run existing 16-bit Windows Sockets applications, Windows NT supplies **winsock.dll**. In addition, Windows NT offers 32-bit Windows Sockets support in the DLL called **wsock32.dll**. In general, all of the Windows Sockets routines in the 32-bit DLL are identical to their 16-bit counterparts, although their parameters are widened to 32-bits.

The most significant difference in programming Windows Sockets applications for Windows NT is that Windows NT is a fully preemptive, multithreaded operating system. Therefore, if an application blocks on a Windows Sockets call, the rest of the system is not negatively impacted. In addition, it is feasible to write a multithreaded application which uses one thread to process user input and another to block on sockets calls. Such an application could use the blocking sockets calls and still be responsive to user input.

The asynchronous Windows Sockets calls are still advantageous in Windows NT. The most significant advantage is that they allow an application to be fully event-driven, fitting better within the Windows programming paradigm. In addition, 32-bit versions of Windows Sockets will soon be available for Win32s. An application written to use the asynchronous routines can be easily ported to Win32s without the negative system impacts of blocking calls.

Transport Independence

We've focused on TCP/IP for most of this article, but we promised to describe how an application can use Windows Sockets other transport protocols in Windows NT and Chicago. Constrained by space, we'll focus on the highlights and the key features that differentiate different transport protocols, and on the mechanisms for opening sockets for use with different transport protocols.

Although these are not explicitly addressed in the Windows Sockets specification, the fact that the fundamental job of a transport protocol is data transfer, along with the fact that Windows Sockets provides a rich interface for low-level data transfer in applications, make the interface to Windows Sockets over different transport protocols quite feasible. In fact, having a common, well-known API interface allows applications to be ported quickly between different transport protocols, and there are several successful examples of this.

Because of the architecture of the Windows Sockets components of Windows NT and Chicago, all supported transports are accessed through the same DLL, *wsock32.dll*. In fact, a single application may simultaneously use sockets of different transport protocols.

An application uses the parameters to the **socket()** API to specify the protocol it desires. For example, a TCP socket is opened with

```
s_tcp = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
```

while an SPX socket is opened with

```
s_spx = socket( AF_IPX, SOCK_SEQPACKET, NSPROTO_SPX );
```

After opening a socket for a specific protocol, all actions performed on the socket go through the specified protocol. In the above example, if the application did a **listen()** on the s_tcp socket but not on the s_spx socket, it would receive incoming connects through TCP but not receive incoming connects through SPX.

Protocol Characteristics

An application which uses different transport protocols must be aware of the different characteristics of transport protocols. While all transport protocols share the same fundamental purpose of bidirectional data transfer, many details of the data transfer differ. Following is a list of key characteristics of transport protocols that Windows Sockets applications need to be aware of.

Addressing

The most obvious difference between different transport protocols is the way in which each uses transport addresses, or "sockaddrs" in Windows Sockets terms. Virtually all transport protocols expose different address formats to applications, the one exception being Netbios protocols which all use the same address format.

A number of Windows Sockets APIs take sockaddrs as input or output parameters, including **bind(), ccept(), sendto(), recvfrom(), getsockname(),** and **getpeername().** For all these APIs a sockaddr is really two arguments: a pointer to the sockaddfr structure itself and a length integer which gives the number of bytes in the sockaddr. For many transport protocols the length of a sockaddr is always the same, but sockaddr lengths may differ between transport protocols. For example, a TCP/IP sockaddr has 16 bytes and an IPX/SPX sockaddr has 14 bytes.

The format of sockaddr structures used by a transport protocol is specified in the *af* or "address family" argument to the **socket()** API. All TCP/IP sockets (including both TCP and UDP sockets) are opened with the AF_INET address family, while IPX/SPX sockets are opened with AF_IPX. Other transport protocols use different address family values as defined in the header file *winsock.h*.

It is up to an application to fill in a sockaddr when **bind()**ing to a local address or **connect()**ing to a remote address, and also to interpret the sockaddrs returned from the **accept()**, **sendto(), recvfrom, getpeername(),** and **getsockname()** APIs.

Connection-Oriented vs. Connectionless

In a connection-oriented transport, applications are required to establish a virtual circuit (sometimes abbreviated to VC) before data transfer can take place. Virtual circuit establishment is asymmetric: one side, the server, must make known to the transport its willingness to receive incoming connections via the **listen()** API. The other side, the client, initiates the circuit with the **connect()** API, and the server can obtain a socket for the circuit with the **accept()** API. Once the circuit is established, data transfer takes place with the **send()** and **recv()** APIs. There is protocol-level activity which results from this circuit establishment, and more protocol activity tears down the circuit when the sockets are closed. TCP and SPX are examples of connection-oriented transport protocols.

In a connectionless transport, there is no circuit establishment required for data transfer. An application only needs to open and bind a socket, after which it may use the **sendto()** and **recvfrom()** APIs to send and receive data. Of course, in order to specify the remote address for sending data or the address from which received data was sent, an application must specify a sockaddr to these routines. UDP and IPX are examples of connectionless transport protocols.

It is possible to use the **connect()** API on sockets opened for connectionless protocols. This is merely an application convenience, allowing the application to use the **send()** and **recv()** APIs, and this does not result in any protocol activity. If a socket is connected in this manner, it will only receive packets sent from the connected address; other packets destined for the socket are silently discarded.

Reliable vs. Unreliable Data Delivery

Most physical networks are inherently unreliable. They can drop data, corrupt it, deliver it out of order, and deliver multiple copies of it. Some transport protocols hide this unreliability from applications by using protocol-level mechanisms to ensure that data is delivered in the correct order, without loss, corruption, or duplication. Other transport protocols do not make these guarantees; they simply make a best effort to get the data to its intended recipient, but physical network failures can lead to data integrity problems.

Because they have less inherent overhead, unreliable protocols often provide better performance, especially for applications which send small amounts of data in a request-response (transaction) format. However, connectionless protocols impose the burden on the application of doing whatever reliability guarantees are necessary, and therefore lead to more complicated applications. Reliable protocols are usually simpler for the application at the expense of a higher cost in system resources and performance.

Typically, connection-oriented protocols are reliable, while connectionless protocols are unreliable. While this is true of all the transport protocols supported by Windows NT, it is not required that these always go hand-in-hand.

Orderly Release

For a connection-oriented transport, there are two ways to terminate a virtual circuit: orderly and abortive. In an orderly release, both sides get a chance to indicate that they have sent all the data they intend to send, and only when both sides are done is the actual circuit terminated. For such a release it is possible to have one side indicate that it is done sending with the **shutdown()** API while the other side continues sending. In an abortive termination of a virtual circuit, one side decides that it is time to terminate the circuit and unilaterally ends the connection. If the other side attempts to use the connected socket, the request fails.

By default, **closesocket()** also attempts an orderly release of the connection. However, under several circumstances, such as pending unreceived, outstanding **send()** calls, and more, a **closesocket()** will result in an abortive close.

All transport protocols support the concept of an abortive release, but not all protocols support orderly release. If an application attempts an orderly release on a transport which does not support orderly release, the connection is terminated abortively. TCP supports orderly release, while SPX supports only abortive release.

Stream-Oriented vs. Message-Oriented

Connection-oriented transport protocols can be either stream-oriented or message-oriented. In a message-oriented transport protocol, each block of data is sent and received as a single block of data. An individual message is never broken down when received, although a transport protocol may be forced to break down a message if it is larger than the physical network's maximum transmission unit (MTU). However, this is transparent to the application.

In a stream-oriented transport, data is sent and received as a continuous stream of data without any message boundaries. The transport protocol is free to coalesce data from discrete **send()** calls and deliver it to a single **recv()** call as well as to break down the data buffer given to a single **send()** call and deliver it to several **recv()** calls. An application has no control over how these breakdowns occur, and an application gets no notification of how data was sent. If an application needs to communicate in discrete messages, it is the responsibility of the application to do message framing, for example by proceeding message data by the length of the message on send and receiving until the specified length has arrived.

Some message-oriented protocols support stream semantics in addition to message semantics. If an application opens a socket with SOCK_STREAM on such a transport, then the transport ignores all message boundaries when transmitting and receiving data. It simply delivers data as it becomes available without regard to whether the data comprises a complete message. This feature is handy for maximizing application portability. Note, however, that stream-oriented transport protocols cannot implement message-oriented semantics due to lack of information.

Expedited Data

Some connection-oriented transport protocols support the concept of expedited data, also referred to as urgent data or out-of-band data. Data sent with the MSG_OOB flag in the **send()** API is sent as expedited data, and the transport protocol makes every effort to deliver the data as quickly as possible. Often the data is delivered out of order so that it gets to the other application as quickly as possible. Applications interested in maximum portability should avoid expedited data since it is not supported by all transport protocols.

Connect and Disconnect Data

A few transport protocols support connect and disconnect data, where a small amount of data is sent by each side when a virtual circuit is established or terminated. This data can be anything, but is often negotiation information about the version of software connecting, etc. Again, applications desiring protocol portability should avoid this feature since it is not universally supported.

Broadcasts

Most connectionless transport protocols support broadcast in the same fashion, where any bound socket can send a broadcast if the SO_BROADCAST option is set, and broadcasts sent to the appropriate local endpoint are received without any additional work on the part of the application. Netbios transports, however, handle broadcasts somewhat differently. In order to receive broadcasts, an application must bind to the Netbios broadcast address, which is an asterisk (*) followed by 15 spaces (ASCII character 0x20). This means two things: a socket must be specially bound to receive broadcasts, and applications can't depend on receiving broadcasts intended only for a specific application, since *all* Netbios broadcasts are delivered to this address. In other protocols such as UDP/IP and IPX, broadcasts are delivered to a socket only if the broadcast was sent to the same port to which the socket was bound.

Guidelines for Transport Independence

The Windows Sockets interfaces of Windows NT and Chicago are designed to fully support all of the transport protocol characteristics listed above. This allows transport-specific applications to fully utilize all the idiosyncrasies of a particular protocol. However, when writing transport-independent applications, it is important to minimize or, preferably, eliminate all uses of features that are not supported by all transport protocols.

Many of the resulting guidelines are fairly obvious:

- Don't send or attempt to receive expedited data. If you must be able to receive expedited data to be compatible with older software, use the SO_OOBINLINE socket option and treat expedited data as normal data in your data reception code.

- Never use connect or disconnect data. If you need to include information at the start or end of a transmission, put it in the normal data stream.

- If you control your application's on-the-wire data format, perform your own message framing for both stream and message protocols. Message framing means to make the length of the message self-describing, for example, by including the message length as the first two bytes of the message. The application can then read two bytes, know the total message size, and read the remainder of the message. This abstracts the distinction between message-oriented and stream-oriented transport protocols.

- Assume that the transport protocol doesn't support a graceful close. It is always possible to do an abortive close, but a graceful close is not always supported. Use an application-level protocol to ensure that all data has been correctly transmitted.

- Typically, both connectionless and connection-oriented transport protocols will be available on any given machine. Therefore, it is reasonable to use whichever is most convenient for your application, which will typically be connection-oriented since it frees the application from having to worry about reliable data delivery.

Where is Windows Sockets Headed?

Although Windows Sockets is a new technology, much evolution has already taken place in the TCP/IP community. Microsoft's new 32-bit operating system, Windows NT, will offer Windows Sockets support for both 16- and 32-bit Windows applications. Microsoft is encouraging third-parties and corporate developers to use Windows Sockets as the client-server and distributed application API by including the Windows Sockets API as part of WOSA, the Windows Open Services Architecture

Several PC-based TCP/IP implementations are offering Windows Sockets support, and over two-dozen application vendors are shipping Windows Sockets-compatible applications. Moreover, many corporations are standardizing on Windows Sockets as the API of choice for client-server networking application development.

The Windows Sockets waters may remain calm for a short while, probably only long enough to allow application vendors and TCP/IP implementors to produce 1.1-compatible offerings. There are already some ideas of what features future revisions of the specification may incorporate: transparent transport independence, access to raw sockets for lower-level network functions, and the ability to share a connected socket between different applications to name a few.

The success of the Windows Sockets effort has spawned two similar undertakings: Windows SNMP and ONC/RPC for Windows. Both of these efforts strive to provide application vendors with a common API across divergent implementations under Microsoft Windows. Many of the participants of these efforts come from the Windows Sockets group, but more interested parties are joining every day.

If Windows Sockets is a sign of things to come in Windows networking, application development under Microsoft Windows will continue to become easier, more flexible, and more powerful.

WormHole – A Sample Application

In order to demonstrate how Windows Sockets might *really* work in a real application, we developed WormHole. WormHole is a peer Windows Sockets application which allows users to establish network "wormholes" between systems and then drag and drop files to one another. The WormHole program is a simple MDI (multiple document interface) application which runs as a 16-bit application on Windows 3.1 systems with a Windows Sockets compliant TCP/IP implementations. WormHole can be conditionally compiled as a 32-bit Windows NT application which will run over NT's 32-bit TCP/IP transport and 32-bit Windows Sockets interface. The makefile explains how to compile this application for Windows NT.

In order to demonstrate as many Windows Sockets concepts as possible, WormHole utilizes both stream and datagram sockets, and is completely event-driven (asynchronous). This allows a WormHole to simultaneously service multiple client connections *and* act as a WormHole client. "Host windows" are created when a user specifies a destination system (either by IP address or hostname) to connect with. Specifying a remote system does *not* establish a network connection, it simply creates a host window on the client to provide feedback during file transfers.

"Wormholes" (connections) are established everytime a user initiates a file transfer by dragging a file from the file manager into a host window. We refer to the system on which this happens as the WormHole client. The WormHole application implements a very simple protocol which would not be entirely uncommon in a production environment. The wormhole setup takes place over a simple pair of datagram frame transactions, followed by stream connection establishment which facilitates the file transfer. The following diagram illustrates the transactions for connection establishment. Remember that since WormHole is a peer application, every instance of WormHole on the network is capable of acting as both a WormHole client and a WormHole server simultaneously.

{ewc msdncd, EWGraphic, bsd23459 8 /a "SDKD.BMP"}

The WormHole Protocol

The wormhole establishment begins on the client side by sending a FILE frame datagram which specifies to the server the name of the file to transfer, the length of the file, and a unique transaction identifier (*xid*). The *xid* allows the server to distinguish between multiple outstanding FILE requests. The server acknowledges the FILE request with either a PORT frame or a NACK frame. A PORT frame informs the client that the server will accept an incoming file, returning a specified stream socket port for the client to connect to as well as the *xid* specified in the original request. If the server wishes to refuse the FILE request, it may respond with a NACK (negative acknowledgment) frame which contains the *xid* it is refusing and optionally an error code (for example, "insufficient disk space", "duplicate filename").

Upon receipt of a PORT frame, the client establishes a stream socket connection to the specified port and begins to transfer the file. Since the server is listening on a port it created in association with a particular transaction, *and* it knows the file name and length, it is not necessary to transfer anything aside from the actual file data. Once the entire file has been received by the server, the stream connection is shut down gracefully.

WormHole Protocol Frame Types and Contents

Datagram Connection Request	Datagram Connection Acknowledge	Datagram Connection Refuse
FILE	PORT	NACK
Transaction ID	Transaction ID	Transaction ID
File length	Port Number	Error code (optional)
File name		

Because the connection request (FILE), acknowledgment (PORT) and refuse (NACK) frames are all transmitted on the network using an unreliable datagram protocol, it is quite possible for these frames to be lost in the shuffle of network activity. This said, the WormHole protocol also implements some retry timers to allow a client to retry a failed connection request. A WormHole client will retry a connection request by retransmitting up to four FILE frames to the server. To keep things simple, the server does not implement a retry timer on possible lost PORT or NACK frames. We instead rely on the client to resend a FILE request in the event that a PORT frame gets lost.

Since the server creates a local window, transaction association, and the like upon receipt of a FILE frame, it does maintain a timer per file transaction to allow for the cleanup of acknowledgment (PORT) frames which go unanswered. Finally, since the file transfer itself takes place over reliable stream sockets, it is not necessary to implement these types of precautions for the actual file transfer. WormHole simply relies on the failure of the *send()* and *recv()* APIs in the event of network or connection problems.

Windows Sockets API Function Overview

Socket Functions

accept()	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
bind()	Assign a local name to an unnamed socket.
closesocket()	Remove a socket descriptor from the per-process object reference table. Only blocks if SO_LINGER is set.
connect()	Initiate a connection on the specified socket.
getpeername()	Retrieve the name of the peer connected to the specified socket descriptor.
getsockname()	Retrieve the current name for the specified socket
getsockopt()	Retrieve options associated with the specified socket descriptor.
htonl()	Convert a 32-bit quantity from host byte order to network byte order.
htons()	Convert a 16-bit quantity from host byte order to network byte order.
inet_addr()	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
inet_ntoa()	Converts an Internet address value to an ASCII string in "." notation i.e. "a.b.c.d".
ioctlsocket()	Provide control for descriptors.
listen()	Listen for incoming connections on a specified socket.
ntohl()	Convert a 32-bit quantity from network byte order to host byte order.
ntohs()	Convert a 16-bit quantity from network byte order to host byte order.
recv()*	Receive data from a connected socket.
recvfrom()*	Receive data from either a connected or unconnected socket.
select()*	Perform synchronous I/O multiplexing.
send()*	Send data to a connected socket.
sendto()*	Send data to either a connected or unconnected socket.
setsockopt()	Store options associated with the specified socket descriptor.
shutdown()	Shut down part of a full-duplex connection.
socket()	Create an endpoint for communication and return a socket descriptor.

**These functions may block if acting on a blocking socket*

Database Functions

gethostbyaddr()*	Retrieve the name(s) and address corresponding to a network address.
gethostname()	Retrieve the name of the local host.
gethostbyname()*	Retrieve the name(s) and address corresponding to a host name.
getprotobyname()*	Retrieve the protocol name and number corresponding to a protocol name.
getprotobynumber()*	Retrieve the protocol name and number corresponding to a protocol number.
getservbyname()*	Retrieve the service name and port corresponding to a service name.
getservbyport()*	Retrieve the service name and port corresponding to a port.

**These functions may block if acting on a blocking socket*

Windows Sockets Asynchronous Extensions

WSAAsyncGetHostByAddr()	A set of functions which provide asynchronous
WSAAsyncGetHostByName()	versions of the standard Berkeley
WSAAsyncGetProtoByName()	getXbyY() functions. For example, the
WSAAsyncGetProtoByNumber()	WSAAsyncGetHostByName() function provides an asynchronous message based
WSAAsyncGetServByName()	implementation of the standard Berkeley
WSAAsyncGetServByPort()	gethostbyname() function.
WSAAsyncSelect()	Perform asynchronous version of select()
WSACancelAsyncRequest()	Cancel an outstanding instance of a WSAAsyncGetXByY() function.
WSACancelBlockingCall()	Cancel an outstanding "blocking" API call
WSACleanup()	Sign off from the underlying Windows Sockets DLL.
WSAGetLastError()	Obtain details of last Windows Sockets API error
WSAIsBlocking()	Determine if the underlying Windows Sockets DLL is already blocking an existing call for this thread
WSASetBlockingHook()	"Hook" the blocking method used by the underlying Windows Sockets implementation
WSASetLastError()	Set the error to be returned by a subsequent WSAGetLastError()
WSAStartup()	Initialize the underlying Windows Sockets DLL.
WSAUnhookBlockingHook()	Restore the original blocking function

Obtaining the Specification

Before you get too excited about writing a Windows Sockets application, it's probably a good idea to get your hands on the specification itself. The current version of the specification is 1.1 and it is distributed electronically on the Internet and Compuserve. If you're not able to obtain the specification electronically, ask a Windows Sockets vendor − they are generally willing to provide you with a copy on disk.

The most recent versions of the specification are available via anonymous FTPon *rhino.microsoft.com* or *sunsite.unc.edu* (in the directory */pub/micro/pc-stuff/ms-windows/winsock*). On Compuserve, check the Microsoft Software Library forum (*go msl*). A variety of different formats are readily available: **.doc** (Microsoft Word), **.wri** (Microsoft Windows Write), **.ps** (PostScript), **.txt** (ASCII text), or **.rtf** (Rich Text

Format). In addition to the specification, you will also find Joel Goldberger's extremely useful Windows Sockets online help file (winsock.hlp), sample applications and other nifty stuff.

Note:

For developers familiar with the 1.0 specification, the Microsoft Word and PostScript versions of the 1.1 specification offer "change-bars" in the left-hand margin to denote changes made to the 1.0 specification.

The Windows Sockets Electronic Discussion List

If you're interested in Windows Sockets, your comments, experience and feedback is welcome. Currently, there is an electronic mailing list of several hundred Windows Sockets developers, users and wizards which offer technical support and discussion of the Windows Sockets API. To subscribe, e-mail *listserv@sunsite.unc.edu*, including as both subject and body the line "SUBSCRIBE WINSOCK <your internet address>". If you're a Usenet user, the mailing list is cross-posted to the *alt.winsock* newsgroup. In addition, you also might want to peruse the *comp.programmer.win-32* and *comp.protocols.tcp-ip.ibmpc* groups which frequently see Windows Sockets-related traffic.