

Microsoft* Windows SDK Knowledge Base: Kernel

Prepared 11/17/93



Kernel APIs



Compiler/Runtime



Debug Kernel



Disk Operations



File I/O



Floating Point



ISRs/TSRs/Interrupts



Memory Management



Memory Models



Tasks/Instances



ToolHelp













WinOLDAp

THE INFORMATION IN THE MICROSOFT KNOWLEDGE BASE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. MICROSOFT DISCLAIMS ALL WARRANTIES EITHER EXPRESSED OR IMPLIED, INCLUDING THE WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL MICROSOFT CORPORATION OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER INCLUDING DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL, LOSS OF BUSINESS PROFITS, OR SPECIAL DAMAGES, EVEN IF MICROSOFT CORPORATION OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES SO THE FORGOING EXCLUSION OR LIMITATION MAY NOT APPLY.



Kernel APIs

-  [INF: Determining the Version of MS-DOS from a Windows App](#)
-  [Applications Must Delete Files Created with GetTempFileName\(\)](#)
-  [INF: Information About Windows Catch and Throw Functions](#)
-  [FIX: FatalAppExit\(\) Function Missing from WINDOWS.H](#)
-  [FIX: SwitchStackBack\(\) Function Causes UAE](#)
-  [PRB: GetModuleHandle and Long Path Causes UAE](#)
-  [INF: Retrieving the Filename of an Application or DLL](#)
-  [FIX: ExitWindows\(\) Returns No wReturnCode to MS-DOS](#)
-  [INF: Additional Documentation for GetDOSEnvironment Function](#)
-  [FIX: FreeModule\(\) Declared Incorrectly in WINDOWS.H](#)



Compiler/Runtime



Debug Kernel



Disk Operations



File I/O



Floating Point



ISRs/TSRs/Interrupts



Memory Management



Memory Models



Tasks/Instances



ToolHelp



WinOLDAp



Kernel APIs



Compiler/Runtime

- [INF: vsprintf\(\) %s Parameters Not Cast to LPSTR](#)
- [INF: vsprintf\(\) Buffer Limit in Windows](#)
- [INF: QuickSort Sample Code for Windows 3.00](#)
- [INF: Compiler Switch Options for Windows Protected Mode Apps](#)
- [INF: Overcoming](#)
- [FIX: Improper Arguments to stat Function Cause UAE](#)
- [INF: Drive and Directory Manipulation in Windows](#)
- [PRWIN9105001: C Run-Time getc\(\) Function Corrupts Data](#)
- [INF: C 6.00 -GW Switch Incompatible with Windows in Real Mode](#)
- [PRWIN9106001: Crash When frexp\(\) in Small or Medium Model DLL](#)
- PRB:**
 - [INF: Implicit Casting by C Compiler Can Cause Problems](#)
 - [INF: Using Near and Far Pointers with C Run-Time Functions](#)
 - [INF: Windows Does Not Support OS/2 Family API Calls](#)
 - [INF: Creating Streamlined Code for Protected Mode Applications](#)
 - [INF: Sample Code Replaces sscanf in DLLs for Windows](#)
 - [INF: Using the Linker /ALIGN Option](#)
 - [PRB: SetEnvironment\(\) Returns Incorrect Values](#)



Debug Kernel



Disk Operations



File I/O



Floating Point



ISRs/TSRs/Interrupts



Memory Management



Memory Models



Tasks/Instances



ToolHelp



WinOLDap



Kernel APIs



Compiler/Runtime



Debug Kernel

- [PRB: OutputDebugString\(\) Comments Section Documentation Error](#)
- [INF: Determine Application Stack Size](#)
- [INF: An Annotated Dr. Watson Log File](#)
- [INF: Programs Crash Accessing AUX Port Under Debug Version](#)
- [PRB: Pointer Functions in MASM Can Hang Real Mode](#)
- [INF: Profiling Time Between OutputDebugString and FatalExit](#)
- [INF: Tracking Unrecoverable Application Errors Without CVW](#)
- [FIX: One Cause of FatalExit 0x0403](#)
- [FIX: One Cause of FatalExit in Debug Enhanced Mode](#)
- [FIX: UAE at Application Load Time Caused by Preload Area Size](#)
- [INF: Redirecting Debugging Information Under Windows 3.0, 3.1](#)
- [PRB: Linker Warning L4000](#)
- [FIX: Bad Extended Error Information After Critical Error](#)
- [PRB: Fatal Exit 0x00FF: MakeProInstance for Current Instance](#)
- [PRB: KRNL386: Unable to Enter Protected Mode](#)
- [PRB: Windows FatalExit 0x0280 Error Caused by FAR WinMain](#)
- [INF: Stack Traces Under Windows 3.1 SDK Debugging Kernel](#)
- [PRB: One Cause of Fatal Exit 0x0140](#)
- [PRB: Strange UAE in Windows 3.00](#)
- [PRB: One Cause of Fatal Exit 0x001A](#)
- [INF: Checking for Invalid Global or Local Handles](#)



Disk Operations



File I/O



Floating Point



ISRs/TSRs/Interrupts



Memory Management



Memory Models




Tasks/Instances



ToolHelp



WinOLDap

 [Windows SDK Knowledge Base: Kernel](#)



Kernel APIs



Compiler/Runtime




Debug Kernel



Disk Operations

 [INF: GetDriveType DRIVE_REMOVEABLE Documentation Error](#)

 [INF: Writing Volume Labels to Floppy and Hard Disks](#)



File I/O



Floating Point



ISRs/TSRs/Interrupts



Memory Management



Memory Models



Tasks/Instances



ToolHelp



WinLDAP

 [Windows SDK Knowledge Base: Kernel](#)



Kernel APIs



Compiler/Runtime

































Debug Kernel



Disk Operations



File I/O

-  [PRB: Windows Applications Cannot Share File Handles](#)
 -  [FIX: sopen\(\) Fails When Called from a Windows DLL](#)
 -  [INF: Limits on the Number of Open Files](#)
 -  [INF: File Manager's Mechanism for Sensing File System Changes](#)
 -  [INF: Windows OpenFile Function vs. C Run-Time](#)
 -  [BUG: sopen\(\) Fails When Called From a Windows DLL](#)
-  [INF: Updating Cached Private Profiles \(.INI Files\)](#)
 -  [INF: Handling Critical Errors in a Windows Application](#)
 -  [INF: Opening Files, Compatibility Mode and Windows](#)
 -  [INF: Using OpenFile with Sharing and Inheritance Bits](#)
 -  [INF: Application Dynamically Links to a DLL Using a Class](#)
 -  [PRB: C Run-Time locking Function Causes Sharing Violations](#)
 -  [INF: File Input/Output for Windows-Based Applications](#)
 -  [INF: LZEXPAND.DLL API Documentation](#)
 -  [INF: No MS-DOS Extended Error Info for Windows File Functions](#)
 -  [PRB: Creating File with Exclusive Access Allows Concurrent Use](#)
 -  [INF: Determining That SHARE Is Loaded Under Microsoft Windows](#)
 -  [FIX: SetHandleCount\(\) Causes UAE or Hang](#)
-  [BUG: OpenFile Function Fails on Novell Temp Drive](#)
 -  [SAMPLE: Reading the Boot Sector of a Drive](#)
 -  [PRB: File Handles Cannot Be Shared Between Programs or DLLs](#)
 -  [INF: Failure to Load Resources When All File Handles Are Used](#)
 -  [INF: Do Not Use the MS-DOS APPEND Utility in Windows](#)
 -  [INF: Incomplete Description of SetErrorMode\(\) Function](#)
 -  [BUG: OpenFile Fails When UNC Server Name Longer than 11 Chars](#)
 -  [INF: Sharing Files with Windows for Workgroups Clients](#)
 -  [PRB: File Attributes/Date/Time Fail to Set on Open File](#)
 -  [INF: The](#)
 -  [INF: How OF_SHARE Modes Affect Opening Files](#)
 -  [INF: Windows Code Module to Delete Files](#)



Floating Point



ISRs/TSRs/Interrupts



Memory Management



Memory Models




Tasks/Instances



ToolHelp



WinOLDAp

 [Windows SDK Knowledge Base: Kernel](#)



Kernel APIs



Compiler/Runtime



Debug Kernel



Disk Operations




File I/O



Floating Point

 [PRB: DLL Function Returns Float or Double Value Incorrectly](#)

 [INF: Applications and the Math Coprocessor Under Windows](#)



ISRs/TSRs/Interrupts



Memory Management



Memory Models




Tasks/Instances



ToolHelp



WinLDAP

 [Windows SDK Knowledge Base: Kernel](#)



Kernel APIs



Compiler/Runtime



Debug Kernel



Disk Operations



File I/O



Floating Point



ISRs/TSRs/Interrupts

 [INF: EMS Support in Windows Version 3.00 and 3.10](#)



Memory Management



Memory Models




Tasks/Instances



ToolHelp



WinOLDAp

 Windows SDK Knowledge Base: Kernel



Kernel APIs



Compiler/Runtime



Debug Kernel



Disk Operations



File I/O

































Floating Point





















ISRs/TSRs/Interrupts



Memory Management

-  [INF: How to Get a Pointer to the Stack](#)
-  [INF: How Windows Resolves Far Calls When Movable Flag Is Used](#)
-  [INF: Global Lock Count Changes in Windows 3.x](#)
-  [INF: WINMEM32 Not Version Dependent](#)
-  [INF: Heap Placement in Memory](#)
-  [INF: Overview of How to Share Memory Between Applications](#)
-  [INF: Accessing Physical Memory Using Kernel Exported Selectors](#)
-  [INF: Minimizing Lock and Unlock Calls in Protected Mode](#)
-  [INF: Real Mode Not Supported by Windows 3.1](#)
-  [INF: Shrinking Heap Space](#)
-  [INF: Speed Differences Between WIN /3, WIN /2, and WIN /R](#)
-  [FIX: GlobalReAlloc\(\) Fails in Enhanced Mode](#)
-  [INF: Validating Local Handles](#)
-  [INF: Allocation Limit on WINMEM32 Global32Alloc\(\) Function](#)
-  [PRB: Segment Was Discardable Under 3.0 Notification](#)
-  [INF: GetCodeInfo\(\) Documented Incorrectly](#)
-  [INF: Implementing Linked Lists with Handles in Windows](#)
-  [INF: Windows 3.1 Standard Mode and the VCPI](#)
-  [INF: Windows Enhanced Mode Allocation Limit 16 MB Minus 64K](#)
-  [INF: XMS Calls Under Windows 3.1](#)
-  [BUG: GlobalPageLock Moves Memory Fixed by GlobalFix](#)
-  [INF: Segment and Handle Limits and Protected Mode Windows](#)
-  [PRB: XMS Version Information in MS-DOS Window Incorrect](#)
-  [INF: Shorthand Notation for Memory Allocation Flags](#)
-  [INF: Appropriate Uses of WINMEM32](#)
-  [INF: What EMS Means to Developers](#)
-  [FIX: GlobalReAlloc\(\) Shrinks >1 MB Block to <1 MB UAE](#)
-  [INF: Corrected WINMEM32.DLL Available in Software Library](#)
-  [INF: Maximizing the Use of Available Memory in Windows](#)
-  [INF: Checksums for Windows Executable Image Files](#)

-  [FIX: Microsoft Windows Page Locks GMEM_FIXED Memory](#)
-  [PRWIN9106004: Memory Allocation in Enhanced Mode Hang or UAE](#)
-  [INF: Information About Headings and Labels in HEAPWALK](#)
-  [INF: Future Direction of WINMEM32](#)
-  [INF: Using Memory Below 1 Megabyte](#)
-  [Sample: Global Heap Functions](#)
-  [INF: DPMI Specification Available from Intel](#)
-  [INF: Using GlobalNotify to Implement Real Mode Virtual Memory](#)
-  [INF: Solving the](#)
-  [INF: Determining Free Memory in Windows Enhanced Mode](#)
-  [INF: Demand Paging MS-DOS Applications](#)
-  [PRB: GlobalUnlock Can Cause Fatal Exit 0x02F0](#)
-  [PRB: Reset A20 Bit Set During DPMI Simulate Interrupt Crash](#)
-  [PRB: Protected-Mode GlobalCompact Return Is Not Free Memory](#)
-  [INF: Windows Applications Should Not Use EMS Memory](#)
-  [INF: Memory Access Methods for Protected Mode Applications](#)
-  [INF: Rules for Using Far Pointers to Memory Objects](#)
-  [INF: GlobalReAlloc\(\) and GMEM_ZEROINIT Clarified](#)



Memory Models




Tasks/Instances



ToolHelp



WinOLDap

 [Windows SDK Knowledge Base: Kernel](#)



Kernel APIs



Compiler/Runtime



Debug Kernel



Disk Operations



File I/O



Floating Point









ISRs/TSRs/Interrupts



Memory Management



Memory Models

-  [INF: Large Model and Windows 3.0 Protected Mode](#)
-  [INF: C Run-Time Functions Can Use Far Pointers in Medium Model](#)
-  [INF: Windows 3.0 Does Not Support Static Data Segments > 64K](#)
-  [INF: Sample Code Unlocks Large-Model Extra Data Segments](#)
-  [INF: Using Large Memory Model, Microsoft C/C++, & Windows 3.1](#)
-  [FIX: Using fputc\(\) or fgetc\(\) in Large Model DLL UAEs](#)



Tasks/Instances



ToolHelp



WinOLDAp

 [Windows SDK Knowledge Base: Kernel](#)



Kernel APIs



Compiler/Runtime



Debug Kernel



Disk Operations



File I/O



Floating Point



ISRs/TSRs/Interrupts


















Memory Management



Memory Models



Tasks/Instances

-  [INF: Differences Between Task Handles and Instance Handles](#)
-  [INF: HANDLEs Returned by GetModuleHandle and LoadLibrary](#)
 -  [INF: Retrieving the Names of Simultaneous Tasks Under Windows](#)
 -  [INF: Heap and Stack Usage Within Windows](#)
 -  [INF: Why WinExec\(\) Returns Error Code 8: Insufficient Memory](#)
 -  [INF: Windows: Nonpreemptive vs. Preemptive Scheduling](#)
 -  [INF: How to Determine When Another Application Has Finished](#)
 -  [INF: The Purpose of WINSTUB in Windows SDK](#)
 -  [INF: Sample Code Spawns Task and Waits for its Termination](#)
 -  [BUG: GetModuleFileName Returns Relative File Path](#)
 -  [PRB: Avoiding](#)
 -  [INF: SpawnAndWait DLL Provides Asynchronous Spawn Function](#)
 -  [FIX: Application with No Exports Crashes Under Windows 3.0](#)
-  [INF: Callback Functions in Multiple Instance Applications](#)
 -  [INF: Passing Modified Environments to Child Processes](#)



ToolHelp



WinOLDap

 [Windows SDK Knowledge Base: Kernel](#)



Kernel APIs



Compiler/Runtime



Debug Kernel



Disk Operations



File I/O



Floating Point



ISRs/TSRs/Interrupts



Memory Management










Memory Models



Tasks/Instances



ToolHelp

-  [INF: Spawn an Application and Wait Sample Code](#)
-  [INF: Using TOOLHELP to Determine Free System Resources](#)
-  [INF: Retrieving Application Exit Code in MS-DOS Window](#)
-  [INF: Sample Windows Application Produces Stack Trace](#)
-  [PRB: Error in the THSAMPLE Sample Application](#)
-  [INF: Chaining NotifyRegister Callbacks Issuing Notifications](#)
-  [INF: Sample Windows Application to Unload DLLs from Memory](#)



WinOLDAp

 [Windows SDK Knowledge Base: Kernel](#)



Kernel APIs



Compiler/Runtime



Debug Kernel



Disk Operations



File I/O



Floating Point



ISRs/TSRs/Interrupts



Memory Management



Memory Models












Tasks/Instances



ToolHelp



WinOLDAp

-  [INF: MS-DOS Application Characteristics Under Windows](#)
-  [FIX: Program Execution Halted Until Key Press](#)
-  [PRB: Activating Full-Screen DOS App from Icon in Enhanced Mode](#)
-  [INF: Calculating Memory Requirements for DOS Applications](#)
-  [INF: Determining Windows Version, Mode from MS-DOS App](#)
-  [INF: Keeping a DOS Window Active Under Standard and Real Mode](#)
-  [INF: Determining What Mode and Version of Windows Is Running](#)
-  [INF: Full-Screen DOS Apps Slow Timer Messages in Enhanced Mode](#)
-  [INF: Requested Contents for Windows Problem Reports](#)

INF: Determining the Version of MS-DOS from a Windows App

Article ID:

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

There are at least three ways for an application developed for Windows version 3.x to determine the version of MS-DOS that is running on the system. Both the first and second methods require placing a few lines of inline assembler code into the application. The second method requires MS-DOS version 5.0. The third method requires changing the GetVersion function prototype in the WINDOWS.H header file distributed with the Microsoft Windows Software Development Kit (SDK) version 3.0 (this change is not necessary if the Microsoft Windows SDK version 3.1 is being used).

Even though it is necessary to modify the Windows 3.0 header file, the third method is the most removed from the hardware and can be considered the most portable. The other two methods assume an underlying Intel 80x86 architecture (or emulation).

More Information:

Method 1

This method requires only a few lines of inline assembler code and a call to the DOS3Call function. The following code fragment demonstrates this technique:

```
VOID FAR PASCAL DOS3Call(VOID);    // Use instead of INT 21h
int nMajor;                        // MS-DOS major version
int nMinor;                        // MS-DOS minor version, revision
int nOEMNumber;                   // OEM serial number
static char szUserMsg[80];        // holds user message

_asm
{
    mov     ax, 0x3000             ; Get MS-DOS version
    call   DOS3Call
    mov     nMajor, al            ; Save major number
    mov     nMinor, ah           ; Save minor version number
    mov     nOEMNumber, bh       ; Save OEM Serial number
}

wsprintf(szUserMsg,
         "Running on MS-DOS %d.%d OEM Serial Number %d",
         nMajor, nMinor, nOEMNumber);
MessageBox(hWnd, szUserMsg, "MS-DOS Version", MB_OK);
```


Method 2

This method requires only a few lines of inline assembler code and a call to the DOS3Call function. Additionally, it requires that MS-DOS version 5.0 is running on the system. While this function does not report the OEM serial number, it does report whether MS-DOS is in ROM or in HMA (the High Memory Area). Also the MS-DOS version returned by this method is unaffected by the SETVER command. The following code fragment demonstrates this technique:

```
#define DOSINROM 0x08
#define DOSINHMA 0x10

int nMajor    = 0;
int nMinor    = 0;
int nRevision = 0;
int nDOSFlag  = 0;
static char msg[120];

_asm
{
    mov     ax, 0x3306        ; Get MS-DOS version
    call   DOS3Call
    mov     nMajor, bl        ; Save major number
    mov     nMinor, bh       ; Save minor version number
    mov     nRevision, dl     ; Revision num in low 3 bits
    mov     nDOSFlag, dh     ; MS-DOS version flags
}

wsprintf(msg,
    "Running on MS-DOS %s %s version %d.%d revision %d ",
    (LPSTR) (nDOSFlag & DOSINROM ? "in ROM " : ""),
    (LPSTR) (nDOSFlag & DOSINHMA ? "in HMA " : ""),
    nMajor, nMinor, (nRevision & 0x03));
MessageBox(hWnd, msg, "MS-DOS Version", MB_OK);
```

Important Note for Methods 1 and 2

The DOS3Call function must be prototyped. In assembly language, the appropriate prototype is as follows:

```
extrn DOS3CALL: far
```

Use the DOS3Call function instead of making a direct call to MS-DOS INT 21h. The DOS3Call function runs a little faster than the equivalent INT 21h call under Windows and it ensures that the interrupt will be handled correctly under Windows.

Method 3

The last method involves making a slight modification to the Windows SDK version 3.0 header file, WINDOWS.H. To retrieve the MS-DOS version, change the return type in the function prototype of the GetVersion function call from a WORD to a DWORD. The modified prototype should resemble the following:

```
DWORD FAR PASCAL GetVersion(void);
```

It is not necessary to make the above change to the version of the WINDOWS.H file included with the Windows SDK version 3.1.

The following code fragment demonstrates how to use the GetVersion function to display both the MS-DOS version and the Windows version numbers:

```
DWORD dwVersion;
char szUserMsg[80];

dwVersion = GetVersion();
wsprintf(szUserMsg,
         "Window version %d.%d. MS-DOS version %d.%d",
         LOBYTE(LOWORD(dwVersion)), HIBYTE(LOWORD(dwVersion)),
         HIBYTE(HIWORD(dwVersion)), LOBYTE(HIWORD(dwVersion)));

MessageBox(hWnd, szUserMsg, "GetVersion", MB_OK);
```

Additional reference words: 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrApiMisc

**Applications Must Delete Files Created with GetTempFileName()
Article ID: Q69753**

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

When it is necessary for an application to create a temporary file, it is highly recommended that the GetTempFileName() function be used to provide the name for the file. Temporary files created by this method must be deleted by the application before the application terminates. Windows will not automatically delete these files on application termination.

More Information:

The "Note" on page 13 of the "Microsoft Windows User's Guide" version 3.0 manual states:

...some applications may create temporary files. These filenames generally begin with a tilde character (~) and end with the .TMP extension.... If you quit Windows as described in the preceding procedure, any temporary files are automatically deleted....

To a Windows programmer, this note may seem to imply that the GetTempFileName() API will create a file that will be automatically deleted when Windows shuts down. This is not true; the application must delete its temporary files.

If each application destroys all its temporary files as it shuts down, from the end user's perspective, the files ARE automatically deleted, because the user does not delete them with the File Manager. However, it is each application, not Windows, that performs the delete.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrApiMisc

INF: Information About Windows Catch and Throw Functions
Article ID: Q11926

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The following information describes what the Windows functions Catch and Throw do.

Catch and Throw are analogous to the C functions setjmp() and longjmp(). They are used to save or restore the current environment (including the state of all system registers and the instruction counter) for Windows.

Additional reference words: 2.x 3.00

KBCategory:

KBSubcategory: KrApiMisc

FIX: FatalAppExit() Function Missing from WINDOWS.H
Article ID: Q70806

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9103014

SYMPTOMS

When creating an application that uses the FatalAppExit function, the application fails to compile.

CAUSE

The prototype for the FatalAppExit function is missing from WINDOWS.H.

RESOLUTION

Microsoft has confirmed this to be a problem in Windows version 3.0. Add the following declaration to the top of each module that uses the FatalAppExit function:

```
void FAR PASCAL FatalAppExit(WORD, LPSTR);
```

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrApiMisc

FIX: SwitchStackBack() Function Causes UAE

Article ID: Q70808

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9103016

SYMPTOMS

Using the SwitchStackBack function causes an unrecoverable application error (UAE).

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. There is no way to work around this problem under Windows 3.0 except to avoid using the SwitchStackTo and SwitchStackBack functions.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrApiMisc

PRB: GetModuleHandle and Long Path Causes UAE

Article ID: Q71148

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

When an application calls the GetModuleHandle function and specifies a fully qualified path longer than 63 characters, an unrecoverable application error (UAE) occurs.

RESOLUTION

In Microsoft Windows version 3.0, the GetModuleHandle function uses a 64 character internal buffer. In Windows 3.1, this buffer has been extended to 128 characters. Specifying a longer string overflows the buffer and causes the UAE.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrApiMisc

INF: Retrieving the Filename of an Application or DLL
Article ID: Q72385

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

In the Microsoft Windows environment, the `GetModuleFileName` function provides the filename of the executable file that corresponds to a given module instance handle or data instance handle.

More Information:

`GetModuleFileName` retrieves the full path of an application or a dynamic-link library (DLL). Specify the instance handle or module handle of the executable `hModule` parameter. The syntax of this function is:

```
int GetModuleFileName(HANDLE hModule, LPSTR lpFilename, int nSize);
```

The `hModule` parameter identifies the module or instance handle, `lpFilename` points to the buffer to receive the file name, and `nSize` specifies the buffer size.

When Windows launches an application, its instance handle is a parameter to `WinMain`. For a DLL, the instance handle is a parameter to `LibMain`. The instance handle is also available through the `GetWindow` function, as follows:

```
// hWnd is a handle to any one of the target
// executable file's windows or controls.
hAppInstance = GetWindowWord(hWnd, GWW_HINSTANCE);

nPathLength = GetModuleFileName(hAppInstance,
                               (LPSTR)szPath, PATH_LENGTH);
```

The `GetModuleHandle` function provides the module handle for a specified module. For example, if Microsoft Excel is loaded, the full path to the Excel executable file is available through the following code:

```
// "EXCEL" can be used instead of "EXCEL.EXE".
hModule = GetModuleHandle("EXCEL");

nPathLength = GetModuleFileName(hModule,
                               (LPSTR)szPath, PATH_LENGTH);
```

Given a handle to a window or control in an application, the `GCW_HMODULE` parameter to the `GetClassWord` function provides the application's module handle.

Additional reference words: 3.00 3.10
KBCategory:
KBSubcategory: KrApiMisc

FIX: ExitWindows() Returns No wReturnCode to MS-DOS

Article ID: Q72461

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9103026

SYMPTOMS

When an application calls the ExitWindows function with the wReturnCode parameter set to a value other than zero, MS-DOS receives an "errorlevel" equal to zero after Windows terminates.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrApiMisc

INF: Additional Documentation for GetDOSEnvironment Function
Article ID: Q89568

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

Add the following comment to the documentation for the GetDOSEnvironment function on page 366 of the Microsoft Windows Software Development Kit (SDK) "Programmer's Reference, Volume 2: Functions" manual:

If the TEMP MS-DOS environment variable points to an invalid directory, the GetDOSEnvironment function removes the TEMP environment variable from the returned environment string and replaces its contents with "x" characters.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrApiMisc

FIX: FreeModule() Declared Incorrectly in WINDOWS.H
Article ID: Q77478

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

In the WINDOWS.H header file, the FreeModule() function is declared to return a BOOL value. This is incorrect. As documented on page 4-144 of the "Microsoft Windows Software Development Kit Reference Volume 1," the correct declaration for FreeModule is as follows:

```
void FreeModule(HANDLE)
```

Microsoft has confirmed this to be a problem in the WINDOWS.H header file for the Windows SDK for Windows 3.0. This problem was corrected in the WINDOWS.H file for the Windows SDK for Windows 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrApiMisc

INF: wsprintf() %s Parameters Not Cast to LPSTR
Article ID: Q64759

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, versions 3.0 and 3.1
-

Unrecoverable application errors (UAEs) can result from improperly using the `wsprintf()` function. Any parameter passed to `wsprintf` that corresponds to an `%s` format string MUST be cast to a LPSTR.

The documentation for `wsprintf()` in the "Microsoft Windows Software Development Kit Reference Volume 1" version 3.0 manual states:

<u>Sequence</u>	<u>Meaning</u>
s	Insert a string argument referenced by a long pointer. The argument corresponding to this sequence MUST be passed as a long pointer (LPSTR).

`Wsprintf()` is a function with a variable number of parameters. Therefore, it must be prototyped using the following C calling convention for a variable number of arguments:

```
int FAR cdecl wsprintf(LPSTR, LPSTR,...);
```

Because the only type information in the prototype describes the output buffer and the format string, the C compiler cannot perform implicit casts on the other parameters at compile time. Normally, when a near pointer (`char *`) is used as an argument to a function requiring a LPSTR, the compiler will implicitly cast the (`char *`) to LPSTR, or (`char far *`).

Because the compiler cannot cast any of the additional parameters, in small and medium model programs, any string pointer that is not explicitly cast FAR will be passed to `wsprintf()` as a near pointer. `wsprintf()` attempts to retrieve a far pointer from the stack, which results in an invalid pointer and an unrecoverable application error.

The following two code fragments show incorrect and correct usage of `%s` fields within `wsprintf()`:

```
//INCORRECT use of a near pointer. Assume small or medium model.
```

```
{
char      sz[30];      //sz is a NEAR pointer.
char      szOut[50];  //szOut is also NEAR
LPSTR     szFar = sz; //szFar is FAR
.
.
.
}
```

```

/*
 * Because it is the output buffer, szOut is implicitly cast to a
 * LPSTR. However, sz is pushed on the stack as a NEAR pointer,
 * which wsprintf will pop as a FAR pointer.
 * This call will cause a UAE.
 */
wsprintf(szOut, "sz=%s", sz);

/*
 * This call will succeed since szFar is already a LPSTR.
 */
wsprintf(szOut, "sz=%s", szFar);
.
.
}

```

//CORRECT--

```

{
char      sz[30];      //sz is a NEAR pointer.
char      szOut[50];  //szOut is also NEAR
LPSTR     szFar = sz; //szFar is FAR
.
.
.
/*
 * Because it is the output buffer, szOut is implicitly cast to a
 * LPSTR. Due to the explicit cast, sz is pushed on the stack
 * as a FAR pointer. This call will succeed.
 */
wsprintf(szOut, "sz=%s", (LPSTR)sz);

/*
 * This call will succeed since szFar is already an LPSTR.
 * The cast is redundant, but it's free insurance.
 */
wsprintf(szOut, "sz=%s", (LPSTR)szFar);
.
.
.
}

```

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrComplrWsprintf

INF: vsprintf() Buffer Limit in Windows

Article ID: Q77255

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.0 and 3.1
 - Microsoft Win32 Software Development Kit for Windows NT version 3.1
-

The `vsprintf(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 `vsprintf` 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.x

KBCategory:

KBSubcategory: KrComplrVsprintf

INF: QuickSort Sample Code for Windows 3.00

Article ID: Q64077

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

QSRTWIN is a file in the Software/Data Library that demonstrates using the qsort() function in the C run-time library in an application developed for Windows version 3.0.

QSRTWIN can be found in the Software/Data Library by searching on the word QSRTWIN, the Q number of this article, or S12662. QSRTWIN was archived using the PKware file-compression utility.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrComplrOthrfunc

INF: Compiler Switch Options for Windows Protected Mode Apps
Article ID: Q68801

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

If an application is always to be run under one of Windows' protected modes (standard mode or 386 enhanced mode), you can greatly simplify the function prolog and epilog code, thus producing smaller and faster application code. The Microsoft C Compiler adds the normal function prolog and epilog to far functions when the -Gw option is specified on the compiler command line. However, this prolog and epilog code is only required by functions that are exported from the application.

The standard prolog and epilog code is required for exported functions so that the compiler will generate code to restore the DS register to the data segment appropriate for the function and to establish a stack frame that Windows recognizes when it inspects and modifies the stack. This process is known as "walking" the stack and takes place whenever memory objects are moved in real mode Windows.

In a protected mode, once the DS register is restored by an exported function, it will not change. Thus, nonexported functions called from within an exported function need not modify the DS register. Similarly, code selectors never change, so Windows will not walk the stack. This eliminates the need for any special stack frame. Therefore, with nonexported functions in protected-mode-only applications, the -Gw switch is not required.

More Information:

Please note that applications compiled without the -Gw switch will NOT run in Windows real mode. In real mode, the DS register may change and Windows may walk the stack at any time. Thus, the special stack frame is required.

The Microsoft C Compiler version 6.00 introduces a new option switch, -GW, that is designed to produce smaller prolog and epilog code for nonexported functions in real mode applications. Unfortunately, using this option switch in Microsoft C version 6.00 causes the compiler to generate incorrect code. Microsoft has confirmed this to be a problem in C version 6.00. We are researching this problem and will post new information here as it becomes available.

Warning: The special stack frame created by the -Gw switch is used by most debugging tools. Prolog code for FAR calls increments the BP register by one when entered, making BP odd. Windows fatal exit processing, stack trace listing in CodeView for Windows, and some profilers depend on the odd/even characteristics of the BP register stored in the stack frame to properly back-trace (walk) the stack. Without a proper function prolog, functions that rely on this will not

work as documented.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrComplrSwitch

INF: Overcoming

Article ID: Q69898

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

Code that is compiled using the Microsoft C compiler's warning level 3 (/W3) or higher and that calls functions through FARPROC pointers causes the C compiler to report warning C4071, "No Function Prototype Given."

CAUSE

Pointers to functions are commonly used when an application explicitly loads dynamic-link libraries (DLLs) through the Windows LoadLibrary function. Function pointers declared with FARPROC do not inherit function prototype information.

RESOLUTION

Modify the function pointers declarations to include function prototype information.

More Information:

The following code sample uses the generic FARPROC far-pointer-to-function data type. Compiling the code with Microsoft C at warning level 3 or higher results in a C4071 warning:

```
FARPROC lpfmErrorProc;  
lpfnErrorProc = GetProcAddress(hModule, MAKEINTRESOURCE(1));  
(*lpfnErrorProc)(hWnd, (LPSTR)"Error Message");
```

However, the following code sample defines custom far-pointer-to-function data types which provide information about the function arguments. This code does not produce the warning:

```
// typedef declarations  
typedef VOID FAR PASCAL FNERRORPROC(HWND, LPSTR);  
typedef FNERRORPROC FAR *LPFNERRORPROC;  
  
// variable declaration  
LPFNERRORPROC lpfmErrorProc;  
  
// variable assignment and indirect function call  
lpfnErrorProc = GetProcAddress(hModule, MAKEINTRESOURCE(1));  
(*lpfnErrorProc)(hWnd, (LPSTR)"Error Message");
```

Additional reference words: 3.00 3.10 SR# G910211-88 MICS3 R3.1

KBCategory:

KBSubcategory: KrComplrOthrfunc

FIX: Improper Arguments to stat Function Cause UAE

Article ID: Q70804

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9103012

SYMPTOMS

If the stat function is not passed the name of a valid file, the application experiences an unrecoverable application error (UAE).

STATUS

Microsoft has confirmed this to be a problem in the C run-time libraries provided with the Windows Software Development Kit version 3.0. This problem has been corrected in the libraries provided with the Microsoft C/C++ Development System for Windows version 7.0.

Additional reference words: 3.00 7.00

KBCategory:

KBSubcategory: KrComplrOthrfunc

INF: Drive and Directory Manipulation in Windows

Article ID: Q71760

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

To get or set the current drive or directory in an application developed for the Microsoft Windows environment, use functions provided by the Microsoft C run-time library. The functions listed below are compatible with Windows and are documented in the "Microsoft C Reference" version 6.0 and in the associated QuickHelp on-line help file. Some of the functions listed below are not available in Microsoft C version 5.1.

Note: Any time an application gets or sets the current drive, it should get or set the current directory.

Function	Use
-----	---
chdir	Changes current working directory.
_chdrive	Changes current drive.
_dos_getdrive	Gets the current default drive, using MS-DOS Interrupt 21h Function 19h.
_dos_setdrive	Sets the default disk drive, using MS-DOS Interrupt 21h Function 0Eh.
getcwd	Gets current working directory.
_getdcwd	Gets current working directory for the specified drive.
_getdrive	Gets the current disk drive.
mkdir	Makes a new directory.
rmdir	Removes a directory.
_searchenv	Searches for a given file on specified paths.

Additional reference words: 3.00 5.10 6.00 retrieve

KBCategory:

KBSubcategory: KrComplrOthrfunc

PRWIN9105001: C Run-Time getc() Function Corrupts Data
Article ID: Q72463

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9105001

SYMPTOMS

When using an application that uses the `getc()` function and a Windows edit control, the application experiences data corruption.

RESOLUTION

The `fgetc()` function works properly and should be used as a substitute.

STATUS

Microsoft has confirmed this to be a problem in the Windows Software Development Kit (SDK) version 3.0. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrComplrOthrfunc

INF: C 6.00 -GW Switch Incompatible with Windows in Real Mode
Article ID: Q72465

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Microsoft C compiler version 6.0 has a -GW switch that is used to generate streamlined prolog and epilog code in applications developed for the Microsoft Windows environment. However, the prolog and epilog code generated by -GW should not be used. Use the -Gw switch instead.

More Information:

An application that runs in real mode must store the value of the DS register on the stack each time it makes a far function call. Calling a far function might cause Windows to move the application's data segment when Windows brings in the called code segment from disk. When the data segment moves, the kernel walks the stack and updates all the saved DS values to the new data segment location. Because the prolog code generated by the -GW switch does not push DS on the stack, when the kernel walks the stack, some other data is on the stack where the stored DS register should be. When the debugging kernel walks the stack and encounters the unanticipated data instead of the stored DS register, it produces a "cannot discard segment" FatalExit error message.

The Microsoft C compiler generates the following assembly language code when the -GW switch is specified:

Prolog	Epilog
-----	-----
inc bp	pop bp
push bp	dec bp
mov bp, sp	ret

To work properly in real mode, the prolog code should be modified to resemble the following:

```
inc bp
push bp
mov bp, sp
push ds      ; save DS value on stack for stack walking routine
```

The Microsoft C compiler generates the modified code when the -Gw switch is specified.

The following information describes the situations when it is necessary to use the -Gw switch:

Applications developed for real mode: Always specify the -Gw switch for real mode. Because Windows can walk the stack at any time, the

Windows prolog and epilog code must be executed for all far functions, whether or not they are exported.

Applications developed for protected (standard or enhanced) mode: The -Gw switch is required in protected mode only during the process of debugging an application when a meaningful stack trace is needed or if the module contains exported functions. In an exported function, the Windows prolog code is required to set the DS register to the correct DGROUP value, even in protected mode. Therefore, the extended prolog code generated by -Gw is required. However, non-exported far functions do not require this code.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrComplrSwitch

PRWIN9106001: Crash When frexp() in Small or Medium Model DLL
Article ID: Q73184

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9106001

SYMPTOMS

Although the C run-time function `frexp()` is included in the Windows version 3.0 libraries for dynamic-link libraries (DLLs), when it is called from a Windows small model or medium model DLL, the application crashes.

STATUS

Microsoft has confirmed this to be a problem in the DLL libraries for Windows version 3.0. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrComplrOthrfunc

PRB:

Article ID: Q74699

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

The linker reports an "export not defined" error even though all the functions listed in the EXPORTS section of the definitions (DEF) file are defined.

CAUSE

One of the exported functions uses the C calling convention (cdecl).

RESOLUTION

In the C calling convention, the case of function names is preserved, and the name of each function is preceded by an underscore. This convention must also be used in the DEF file. For example, the function declaration

```
int FAR Function()
```

is exported as:

```
_Function
```

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrComplrSwitch

INF: Implicit Casting by C Compiler Can Cause Problems

Article ID: Q74739

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

When a function call is made in ANSI C, the compiler implicitly casts the arguments passed to the function to the types specified in the function's prototype. Implicit casting to promote signed integers or characters (int, char) to longer unsigned types (DWORD, WORD) can cause unexpected behavior. The difficulties occur because the signed shorter value is promoted by extending its sign bit to the high-order bits of the unsigned longer type.

An application can avert the problems caused by sign extension by explicitly casting function arguments to unsigned short types.

More Information:

In accordance with the ANSI standard, if the shorter value has the sign bit set, the compiler first converts the value to a signed longer value by extending the sign. The compiler extends the sign by filling the high-order bits with 1s. It then converts the signed longer value to unsigned by adding to it the number that is one larger than the largest unsigned value of that type. This does not change the bit pattern in a 2s complement implementation. For more information, see Section 3.2.1.2 of the ANSI C Standard.

To see how this can cause unexpected behavior, consider an application in the Microsoft Windows graphical environment that calls the GlobalAlloc function. The second parameter of the function, dwBytes, is an unsigned long quantity. However, in this application, this parameter contains a signed integer expression that evaluates to a number greater than the largest positive signed integer value (32,767):

```
HANDLE FAR PASCAL GlobalAlloc(WORD, DWORD); // function prototype

int a, b; // int = short (16-bit) signed integer

a = 9500;
b = 4;

GlobalAlloc(GMEM_MOVEABLE, a*b);
```

The result of a*b is 38,000 (1001010001110000), and the sign bit of the int is set. To implement the implicit cast to an unsigned long value (DWORD), the value is first converted to a signed long value:

```
111111111111111111111111001010001110000
```

The value that is one greater than the largest unsigned long value is then added, as follows:

```
11111111111111111111001010001110000
+ 1000000000000000000000000000000000
-----
1111111111111111111100101000111000 (4,294,939,760 decimal)
```

GlobalAlloc attempts to allocate 4,294,939,760 bytes of memory rather than 38,000, and it fails. The GlobalAlloc call in the application should be as follows:

```
GlobalAlloc(GMEM_MOVEABLE, (WORD)a*(WORD)b);
```

Problems caused by implicit casting and sign extension are also encountered frequently when an application passes characters to the AnsiUpper and AnsiLower functions. The prototypes for these functions are as follows:

```
LPSTR FAR PASCAL AnsiUpper(LPSTR);
LPSTR FAR PASCAL AnsiLower(LPSTR);
```

To pass a signed character to AnsiUpper,

```
AnsiUpper((DWORD)(BYTE)c) is correct,
AnsiUpper((DWORD)c) is incorrect.
```

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrComplrSwitch

INF: Using Near and Far Pointers with C Run-Time Functions
Article ID: Q74788

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In an application designed for the Microsoft Windows graphical environment, many C run-time functions do not work with memory allocated by the GlobalAlloc function when the application is developed in the small or medium memory model.

More Information:

MS-DOS (non-Windows) applications written in the small or medium memory model assume the presence of only one data segment (DS). Therefore, the C run-time functions assume that DS will not change.

However, an application can store data in a block of memory allocated with the GlobalAlloc function and locked with the GlobalLock function. The segment returned from GlobalLock will be different from the application's data segment. Specifying the alternate data segment in a C run-time function that assumes a near pointer results in the following C compiler warning:

WARNING: Segment Lost in Conversion

For example, the following code passes far pointers to a run-time function incorrectly:

```
hMem = GlobalAlloc(...);  
lpMem = GlobalLock(hMem);  
  
strcpy(szBuffer, lpMem);  
  
GlobalUnlock(hMem);
```

This section of incorrect code produces one of two results.

1. If the offset of lpMem extends past the end of application's data segment (DS), the application experiences an unrecoverable application error (UAE).
2. The function copies information from some random portion of the application's DS into the buffer.

If the following line of code is used, the function overwrites data in the application's data segment, which causes the application to crash or run incorrectly:

```
strcpy(lpMem, szBuffer);
```

Four ways to work around this situation are:

1. For the most common C run-time functions, Windows provides equivalent functions that use far pointers. These functions include:

- lstrcat
- lstrcmp
- lstrcmpi
- lstrcpy
- lstrlen

2. Use the far pointer versions of these functions (`_fstrcat`, `_fstrcmp`, and so on) provided by the Microsoft C Optimizing Compiler versions 6.0 and later.
3. For the less common C run-time functions, write a far-pointer version as part of the application code. Most of the Microsoft C run-time library source code is available from Microsoft.
4. Use the large memory model. However, using the large model in an application for Windows has many disadvantages and doing so is not encouraged.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrComplrOthrfunc

INF: Windows Does Not Support OS/2 Family API Calls

Article ID: Q43052

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

Windows does not support applications coded with OS/2 Family API (FAPI) calls.

More Information:

FAPI works in the following manner:

1. The code contains references to OS/2 FAPI calls (such as DosOpen, DosRead, and so on).
2. When the program is loaded into OS/2 protected mode, the system loader dynamically links the FAPI calls to the OS/2 system-services DLLs (Dynamic-Link Libraries).
3. When the program is loaded into real mode (MS-DOS or OS/2 compatibility box), what actually gets loaded is a small program called the "FAPI Loader and Linker." It is this program that loads the real code; it dynamically links the FAPI calls to a special library of support routines that translate FAPI calls into 80x86 code and MS-DOS interrupts (INT 21H Function xx).

This process is also described on Page 251 of Gordon Letwin's book titled "Inside OS/2" (Microsoft Press, 1988).

FAPI works well for programs that need to run in MS-DOS and OS/2 protected mode. The problem is that Windows uses the "New EXE Format" for programs, bypassing the standard MS-DOS entry point. For example, if a Windows program is run outside of Windows (in MS-DOS), the following message appears, and the program terminates:

This program requires Microsoft Windows

MS-DOS is not responsible for this message; the Windows program itself is responsible. The way the Windows program works is very similar to OS/2: it uses dual entry points into the .EXE file. In MS-DOS, a short program that prints the above message runs; however, in Windows, a true Windows application runs using the other entry point in the .EXE file.

Therefore, the problem is narrowed down to the following: if the FAPI Loader and Linker program is run using the standard MS-DOS .EXE file entry point, and Windows starts an application using a different entry point, the dynamic linking of the FAPI routines will not occur.

Therefore, FAPI calls cannot be used in Windows applications.

To avoid this problem, do the following:

Instead of using low-level DOS calls (INT 21H Function xx) in a Windows application and OS/2 API calls (DosRead, DosOpen, and so on) in a Presentation Manager (PM) application, use the C run-time I/O routines for all of these applications.

This will work because the Microsoft C Compiler and the run-time libraries supply versions of the libraries that work in both OS/2 and MS-DOS. By moving C code to PM and to Windows, it will not be necessary to rewrite it, and the appropriate conversion routines will be supplied at link time.

Be sure not to use high-level (stream) I/O routines in C with Windows; just use the low-level (handles) versions. This topic is discussed in Charles Petzold's book titled "Programming Windows" (Microsoft Press, 1988), and in other articles in the Microsoft Knowledge Base. For more information, query on the following words:

prod(winsdk) and c and low and level

Additional reference words: SR# G890110-9309 2.x 2.03 2.1 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrComplrSwitch

INF: Creating Streamlined Code for Protected Mode Applications
Article ID: Q75253

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

If an application will be run only under one of Windows' protected modes (standard mode or enhanced mode), and it is acceptable to allow only one instance of the application to run at any given time, there are techniques that can be used to create smaller, faster application code. This article details the techniques and the factors that must be considered when they are used.

More Information:

In Windows real mode, when Windows brings a portion of code in from disk, other objects in memory can be moved about to make room. Every FAR call (a call to another code segment) and every access to a resource (menu, dialog box template, string table, and so forth) can cause memory to move.

Windows allocates a data segment for each instance of each application run. These data segments are objects that can be moved about as code and resources are accessed.

Windows provides a data segment identifier to the application. This value is placed into the DS register. The function prolog code, generated by the C compiler when the -Gw switch is specified, stores the value of the DS register on to the stack before any other function code is executed. If the function changes the value of DS, the function epilog code will restore the original value when the function exits.

When Windows moves memory, it "walks" the stack, looking for the stored DS values. Windows updates the stored information to reflect the new location for the data segment.

In contrast, under Windows' protected modes, the identifier for the data segment does not change, even when memory moves. However, because functions may change the DS register, it is necessary to ensure that the DS register contains the correct value. The Microsoft C Compiler, versions 5.1 and later, provides the `_loadds` keyword, which instructs the compiler to generate prolog and epilog code for a particular function. If an application will be run only in protected mode, the -Gw switch is not necessary. However, the `_loadds` modifier must be specified for every exported function.

The restriction to a single instance is important. The code generated by `_loadds` sets the DS register to point to the data segment of the first instance of the program. The second and subsequent instances

would interfere with each other and with the first instance of the application. The restriction to protected mode is of similar importance. In the absence of prolog and epilog code, if the data segment moved, the value of the DS register would not be updated to reflect the new value; the application would use information in a random portion of memory.

To prevent an application from running in real mode, use the Resource Compiler `-t` option when binding the RES file to the EXE file.

The following code will prevent a second application of the application from running. When an attempt is made to create a second instance of the application, the first instance is brought to the top and activated.

```
if (!hPrevInstance)
{
    /* perform normal RegisterClass processing here */
}
else // This only allows one instance
{
    HWND hWnd, hWnd1;

    hWnd = FindWindow(szAppName, NULL);

    if (IsWindow(hWnd))
    {
        hWnd1 = GetLastActivePopup(hWnd);

        if (IsWindow(hWnd1))
            hWnd = hWnd1;

        BringWindowToTop(hWnd);

        if (IsIconic(hWnd))
            ShowWindow(hWnd, SW_RESTORE);
        SetFocus(hWnd);
    }
    return FALSE;
}
```

Optionally, the `-G2` switch can be specified to generate smaller and faster code for the 80286 or later processors. Because the 8086 processor does not support protected mode, this does not exclude any other equipment.

To summarize, an application that will run only in Windows protected mode does not require the Windows prolog and epilog code. Applications that are compiled without the C compiler `-Gw` switch must have the following three attributes:

1. The `_loadds` modifier is specified for all exported functions.
2. The Resource Compiler `-t` switch is specified to prevent the application from running in real mode.
3. Only one instance of the application is allowed. Use the code

provided above to enforce this restriction.

Additional reference words: 3.0 3.00

KBCategory:

KBSubcategory: KrComplrSwitch

INF: Sample Code Replaces sscanf in DLLs for Windows

Article ID: Q76684

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The Microsoft C run-time library function `sscanf` is not compatible for use with a dynamic-link library (DLL) for the Microsoft Windows graphical environment compiled for the small or medium memory model. The `sscanf` function relies on near pointers in these memory models and fails when the stack segment (SS) and data segment (DS) are not the same.

WSSCANF is a file in the Software/Data Library that can serve as a limited replacement for this function. WSSCANF can be found in the Software/Data Library by searching on the word WSSCANF, the Q number of this article, or S13183. WSSCANF was archived using the PKware file-compression utility.

More Information:

The `sscanf`, `fprintf`, and `scanf` functions are not available in small or medium model DLLs for the Windows environment.

There are two factors that cause these functions to be incompatible:

1. These functions rely on near pointers.
2. These functions expect `SS == DS`.

Because neither of these conditions is true when a function in a DLL uses data from an application, these functions are not available.

The WSSCANF file in the Software/Data contains the source code to a `wsscanf` function that can serve as a limited replacement for the `sscanf` function. The `wsscanf` code is based on the `sscanf` source code in the Microsoft C run-time library. The source code has been modified to work correctly in a DLL, and requires that all parameters are specified as FAR pointers. The following code demonstrates using the `wsscanf` function:

```
char    szBuf[] = "1 3 b000:0200";
int     nValue1, nValue2;
LPSTR  lpPtr;

wsscanf(szBuf, "%d %d %p", (int FAR *)&nValue1,
        (int FAR *)&nValue2, (LPSTR FAR *)&lpPtr);
```

Note that the first two parameters are not explicitly typecast in the function call. The function prototype typecasts the first two

parameters automatically; however, the application must typecast each subsequent parameter. If the application does not typecast each parameter, when the application calls wsscanf an unrecoverable application error (UAE) occurs.

The wsscanf function does not support floating point numbers (the %f, %g, and %e format specifiers).

Additional reference words: 3.00 softlib WSSCANF.ZIP

KBCategory:

KBSubcategory: KrComplrdsss

INF: Using the Linker /ALIGN Option

Article ID: Q47493

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

This article discusses various considerations regarding the Microsoft Linker and its /ALIGN option when it is used to develop applications for the Microsoft Windows environment. This information also applies to the development of dynamic-link libraries (DLLs) for Windows.

More Information:

According to its definition, the Microsoft Linker /ALIGN option "directs the linker to align segment data in the executable file along the boundaries specified by 'size.'" The "size" parameter is in bytes and must be a power of 2. Specifying /ALIGN:16 on the LINK line aligns segments on 16-byte boundaries. Making an /ALIGN:16 specification is recommended for Windows applications and dynamic-link libraries (DLLs) because the default alignment is 512.

When the linker creates an EXE file and /ALIGN:16 is specified, if a segment does not end on a 16-byte boundary, the segment is padded with extra bytes. The next segment always begins at a 16-byte boundary.

If an application contains several small segments, and no /ALIGN option is specified on the Linker command line, each segment will contain a great deal of wasted space and the resulting EXE file will be unnecessarily large. The amount of wasted space is computed as follows:

$$\text{waste} = \text{align} - (\text{segment modulo align})$$

Therefore, for a 514-byte segment, an align size of 512 causes 510 bytes to be wasted. However, for the same segment, an alignment size of 2 bytes does not waste any space.

Problems can arise when the Linker creates a very large EXE file using a small align value because the size of the EXE may exceed the range of values that can be represented by the EXE header segment table.

To demonstrate the problems that can arise, consider a very large EXE file that is linked with an align size of 2. During the process of creating this EXE file, the Linker puts segment 42 at file offset 380,000 and records the position in the New EXE Segment Table. The format of this table is as follows:

Offset	Length	Contents
-----	-----	-----

0h	2	Offset of segment relative to beginning of file after shifting value left by alignment shift count.
2h	2	Length of segment (0h for segment of 65536 bytes).
4h	2	Segment flag word.
6h	2	Minimum allocation size for segment; that is, amount of space Windows reserves in memory for the segment (0h for minimum allocation size of 65536 bytes).

In this case, the offset of the segment to place in the table is $(380,000 \gg 1) = 190,000$, which is too large to store in a 16-bit word (the maximum value is 65,535). Therefore, 58,928 (the low-order 16 bits of 190,000) is stored. Unfortunately, the Linker does not provide any warning of the data loss involved with this step.

When Windows loads segment 42 from the EXE file, it takes the value 58,928 and multiplies it by the align size (2), which results in an offset of 117,856 and does not lead to the desired segment in the file.

For more information on the new EXE (New Executable) file header format, see appendix K (pages 1488-1497) of the "MS-DOS Encyclopedia" (Microsoft Press).

Additional reference words: 2.03 2.10 3.00 2.x

KBCategory:

KBSubcategory: KrComplrSwitch

PRB: SetEnvironment() Returns Incorrect Values

Article ID: Q76590

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

SetEnvironment() returns unexpected values when used to delete an environment.

CAUSE

SetEnvironment() returns 1 when an environment is successfully deleted, not -1 as documented in the "Windows Software Development Kit Reference Volume 1" for version 3.0.

STATUS

The documentation is incorrect.

More Information:

SetEnvironment loads 1 only into al, not ax; therefore, upon return from a successful delete, the high portion of the returned int (short) can contain a random value, depending on how many bytes SetEnvironment copied. SetEnvironment returns 0 (zero) upon failure, as documented.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrComplrOthrfunc

PRB: OutputDebugString() Comments Section Documentation Error
Article ID: Q61106

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

The OutputDebugString() function is documented incorrectly on page 4-326 of the "Microsoft Software Development Kit Reference Volume 1" for version 3.0. The "Comments" section incorrectly states that this function is available only in the debugging version of Windows.

RESOLUTION/STATUS

The documentation should state that OutputDebugString() messages are displayed under both versions of Windows.

Messages that should be written only for debugging purposes should be placed in a conditional compilation block, such as the following:

```
#ifdef DEBUG
    OutputDebugString(lpMessage);
#endif
```

Microsoft has confirmed that this error occurs on page 4-326 of the "Microsoft Software Development Kit Reference Volume 1" for version 3.0. We will post new information here when the documentation has been corrected.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrDebugDebugver

INF: Determine Application Stack Size

Article ID: Q40060

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

During the process of creating an application, it may be useful to determine how much stack space the application requires.

STACK is a file in the Software/Data Library that contains two functions written in Microsoft Macro Assembler (MASM) code to provide this information. An application that displays this data about itself is also included.

STACK can be found in the Software/Data Library by searching on the word STACK, the Q number of this article, or S12169. STACK was archived using the PKware file-compression utility.

Additional reference words: 2.00 2.10 3.00

KBCategory:

KBSubcategory: KrDebugStacktr

INF: An Annotated Dr. Watson Log File

Article ID: Q81142

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

Dr. Watson is a utility included with Windows 3.1 that logs information about applications that fail. This article presents a sample log file and explains what the various entries signify. Comments in the log file start with pound sign (#).

This article annotates a log file for the BICHO application, which tests Windows by faulting in the various ways. Bicho is Bolivian slang for "bug" or "critter."

More Information:

Dr. Watson writes the following line each time it starts execution
unless the [dr. watson] section of the WIN.INI file contains the
line SkipInfo=time:

Start Dr. Watson 0.80 - Thu Sep 26 10:51:28 1991

These lines mark the beginning of a Dr. Watson failure report
They report the version of Dr. Watson and the date and time of the
reported event.

Dr. Watson 0.80 Failure Report - Thu Sep 26 10:51:36 1991

The next line reports that an application named "BICHO" encountered
an "Exceed Segment Bounds" fault while reading memory. The precise
point of failure was also in BICHO, 0x6b bytes after the start of
the DoCommand function.

BICHO had a 'Exceed Segment Bounds (Read)' fault at BICHO
_DoCommand+006b

The following line repeats the previous information formatted for
automatic parsing code. It also includes the instruction that caused
the fault (a push instruction in this case).

\$tag\$BICHO\$Exceed Segment Bounds (Read)\$BICHO _DoCommand+006b\$push
word ptr [fffe]\$Thu Sep 26 10:51:36 1991

The following lines report the contents of the CPU registers:

CPU Registers (regs)

The 16-bit registers are listed first. This information can be
useful to determine what address an instruction modified when the
fault occurred.

ax=1e54 bx=0014 cx=0d7f dx=0111 si=1e54 di=0111

The next items are the instruction pointer (otherwise known as the
program counter), stack pointer, and base pointer. This line also
lists the state of the flag bits. In this example, the Overflow,
Direction, Sign, Zero, and Carry bits are Clear (0), while the
Interrupt, Auxcarry, and Parity bits are Set (1).

ip=02fd sp=230c bp=237a O- D- I+ S- Z- A+ P+ C-

The code segment selector is 0e57, linear address is 8059fbc0. The
code segment's limit is 83f. (Enhanced mode linear addresses often
start with 8xxx.) Accessing a code or data segment beyond its limit
is a common cause of GP faults.

cs = 0e57 8059fbc0:083f Code Ex/R

The next line provides information about the stack segment selector.

ss = 0d7f 8059d5e0:25df Data R/W

The following line provides information about the data segment
selector. Note that the limit is 25df, while the application
attempted to read the value at fffe, which is beyond the segment's
limit.

ds = 0d7f 8059d5e0:25df Data R/W

The following line provides information about the extra segment
selector:

es = 0d7f 8059d5e0:25df Data R/W

The next lines provide information about the 32-bit registers.
If a selector is 0, it corresponds to the null pointer. Attempting
to use a null pointer is another common cause of GP faults.

CPU 32 bit Registers (32bit)

eax = 00001e54 ebx = 00000014 ecx = ffff0d7f edx = 00000111
esi = 00001e54 edi = 00000111 ebp = 0000237a esp = 800422fc
fs = 0000 0:0000 Null Ptr
gs = 0000 0:0000 Null Ptr
eflag = 00000002

The next lines provide information about the Windows installation.

```
System Info (info)
Windows version 3.10
Debug build # The debug version of windows (from the SDK) was running
Windows Build 3.1.048 # This is a prerelease build of Windows, #48
Username Unknown User # Your Name Here
Organization Unknown Organization # Your Organization Here

System Free Space 7131008
```

The following provides the stack size for the current task:

```
Stack base 1122, top 9164, lowest 7504, size 8042
```

Dr. Watson records some statistics about the Windows environment:

```
System resources: USER: 87% free, seg 0777 GDI: 85% free, seg 05d7
LargestFree 6594560, MaxPagesAvail 1610, MaxPagesLockable 267
TotalLinear 1948, TotalUnlockedPages 274, FreePages 52
TotalPages 614, FreeLinearSpace 1611, SwapFilePages 7158
Page Size 4096
4 tasks executing.
WinFlags -
  Math coprocessor
  80386 or 80386 SX
  Enhanced mode
  Protect mode
```

The following records the contents of the stack to determine what
code called the routine that failed:

Stack Dump (stack)

Stack frame 0 indicates that the failure occurred in BICHO, 0x6b
bytes after the start of the DoCommand function, as reported
earlier.

```
Stack Frame 0 is BICHO _DoCommand+006b          ss:bp 0d7f:237a
```

The offending instruction is disassembled in context, as follows:

```
0e57:02f0 e9 02b9          jmp     near 05ac
0e57:02f3 6a 00           push   00
0e57:02f5 9a 8db0 0477    callf  0477:8db0
0e57:02fa e9 02af          jmp     near 05ac
(BICHO: _DoCommand+006b)
0e57:02fd ff 36 fffe      push   word ptr [ffff]
0e57:0301 68 0110         push   0110
0e57:0304 e8 fe5d         call   near 0164
0e57:0307 83 c4 04        add    sp, 04
```

The application tried to read a value from memory at address DS:ffff
and to push that value on the stack. However, the limit of the DS
segment is 25df. The next stack frame documents that the BICHO
application MainWndProc called DoCommand:

Stack Frame 1 is BICHO MAINWNDPROC+0027 ss:bp 0d7f:2388

```
0e57:0670  eb 16                jmp     short 0688
0e57:0672  ff 76 0a            push   word ptr [bp+0a]
0e57:0675  56                 push   si
0e57:0676  e8 fc19            call   near 0292
(BICHO:MAINWNDPROC+0027)
0e57:0679  83 c4 04            add    sp, 04
0e57:067c  99                 cwd
0e57:067d  eb 1f                jmp     short 069e
0e57:067f  6a 00              push   00
```

"USER" in the next stack frame is the Windows module USER.EXE. It
calls application window and dialog procedures. In this case, USER
called the BICHO application's MainWndProc.

Stack Frame 2 is USER IDISPATCHMESSAGE+007e ss:bp 0d7f:239e

In the next stack frame, the BICHO application's WinMain function
called DispatchMessage, which called MainWndProc.

Stack Frame 3 is BICHO WINMAIN+0050 ss:bp 0d7f:23bc

In the last stack frame, the Windows start-up code calls the
application's WinMain function.

Stack Frame 4 is BICHO 1:00a3 ss:bp 0d7f:23ca

The next lines list all the tasks running in the system when the
fault occurred. Dr. Watson itself, the shell application, and the
faulting application will always be included.

System Tasks (tasks)

```
Task WINEXIT, Handle 0daf, Flags 0001, Info 9248 08-09-90 16:52
  FileName C:\MS\WIN\DON\WINEXIT.EXE
Task DRWATSON, Handle 0ea7, Flags 0001, Info 26256 09-23-91 12:00
  FileName C:\WIN31\DRWATSON.EXE
Task PROGMAN, Handle 060f, Flags 0001, Info 110224 09-23-91 12:02
  FileName C:\WIN31\PROGMAN.EXE
Task BICHO, Handle 0da7, Flags 0001, Info 16537 09-11-91 8:45
  FileName D:\BICHO.EXE
```

The last part of a failure report is any information typed in the
"Dr. Watson's Clues" dialog box.

1> I ran a test app that accessed a value
2> beyond the limits of the segment bounds.

Dr. Watson writes this line when it shuts down.

Stop Dr. Watson 0.80 - Thu Sep 26 10:52:10 1991

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrDebugDrwatson

INF: Programs Crash Accessing AUX Port Under Debug Version
Article ID: Q32322

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Some programs that run properly under retail Windows crash when trying to access the AUX port if run under the debug version of Windows. This is because, under the debug version of Windows, the Kernel, User, and GDI do much more extensive error checking. If an error is found, the Kernel dumps the error and a stack trace to the AUX port. This causes an error if the port does not exist.

To run the debugging Kernel, there must be a terminal or other machine connected to COM1. The command, "MODE COM1:9600,N,8,1", (or whatever settings are appropriate) must then be executed before entering Windows. This command can be placed in the AUTOEXEC.BAT file. To test if the debugging Kernel is working properly, enter "dir > com1".

Alternatively, this can be done on a monochrome display by putting OX.SYS in the CONFIG.SYS file and using:

```
symdeb /m .....
```

OX.SYS redirects all I/O for the AUX port to the main keyboard and to the monochrome display. OX.SYS can be found in the Software/Data Library by searching on the keyword OX, the Q number of this article, or S12005. OX was archived using the PKware file-compression utility.

In Windows version 3.0, the WDEB386 debugger can be used with the debug version of Windows. For more information on WDEB386, query on the following words:

```
prod(winddk) and wdeb386
```

Additional reference words: 2.03 2.10 3.00

KBCategory:

KBSubcategory: KrDebugMisc

PRB: Pointer Functions in MASM Can Hang Real Mode

Article ID: Q68537

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

SYMPTOMS

Windows applications written using the Microsoft Macro Assembler (MASM) can fail when run under real mode if pointers to FAR functions are used in the code.

CAUSE

This failure is caused by the following sequence of events:

1. MASM breaks a FAR pointer into separate segment and offset fixup records in the OBJ file.
2. The linker incorrectly resolves the offset record to point to the wrong location in Windows call thunk table.
3. When the FAR function is called through the thunk table, invalid code is executed. This hangs the system.

RESOLUTION

Define a double-word variable in the application's data segment and initialize it to the function address. Instead of using the function pointer directly in any code, use the value stored in the variable. MASM will correctly create a FAR pointer fixup record for the variable. This record is handled entirely by the loader and results in correct operation.

More Information:

Normally, when a FAR function is referenced, a fixup record for the function pointer is placed in the EXE file. The fixup record is resolved at load time by the Windows Kernel to point to a call thunk. A call thunk is a short piece of code used in Windows real mode to check if the code segment containing the called function is currently in memory, and to load it from disk if necessary.

When a FAR function is used in an assembly program as a separate segment and offset, MASM creates two fixup records: a segment that is resolved by the loader and an offset, which is incorrectly resolved by the linker to point to the incorrect offset in the call thunk table.

For example, an assembly program may contain a function pointer reference in the form:

```
;  
; Assume wc is a WNDCLASS structure, and MAINWNDPROC is the  
; main window procedure for the application.  
;
```

```
mov     WORD PTR wc.clsLpfnWndProc, OFFSET MAINWNDPROC
mov     WORD PTR wc.clsLpfnWndProc+2, SEG MAINWNDPROC
```

Since there are two separate references to the function, MASM generates two separate fixup records. The value for OFFSET MAINWNDPROC is incorrectly resolved by the linker.

To generate the correct fixup record, it is necessary to create a variable in the application's data segment that references the function. Always use that variable to load memory or registers with the function address.

```

;
; Define a pointer variable and initialize to the function
; address.
;
data segment
    var_MAINWNDPROC dd     MAINWNDPROC
data ends

...

;
; Make all references to the function through the variable.
;

mov     WORD PTR wc.clsLpfnWndProc, var_MAINWNDPROC.0
mov     WORD PTR wc.clsLpfnWndProc+2, var_MAINWNDPROC.2
```

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrDebugMisc

INF: Profiling Time Between OutputDebugString and FatalExit

Article ID: Q68624

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

During the process of profiling an application, SHOWHITS.EXE typically reveals that a fairly large percentage of time was spent in the Microsoft Windows kernel between the FatalExit and OutputDebugString functions.

This phenomenon occurs because a large amount of code between FatalExit and OutputDebugString is not public; however, this code is common to many public entry points. Therefore, when SHOWHITS.EXE compares the data generated from profiling an application to the data in the kernel symbol file, it matches the closest public symbol to the recorded information. Because the nonpublic code is present, SHOWHITS.EXE reports that the closest public symbols are FatalExit and OutputDebugString. However, the application may not be spending any time in either of these functions; instead, it may be spending time in some private function that is located between these functions.

More Information:

Because of the nonpublic code, determining exactly how much time the application is spending in a particular block of code can be difficult.

The GetCurrentTime function and conditional compilation can be very helpful when timing specific sections of application code. GetCurrentTime returns the amount of time that has elapsed since Windows started. If an application calls this function before and after a particular block of code, the application can generate statistics on how much time is required for this block of code to execute.

The following code demonstrates this idea:

```
// This preprocessor variable should be defined to build a profiling
// version of the application.
#define PROFILING

// This section should be either in the global variable section of the
// application or in the variable declaration section of the function
// to be profiled.
#ifdef PROFILING
    static char szProfBuf[80];
    static DWORD dwPrevTime, dwCurTime;
#endif

// This code initializes the variable and should be placed just before
```

```
// the code that is used to time a block of code.
#ifdef PROFILING
    dwPrevTime = GetCurrentTime();
#endif

// This block of code is placed after each section of code timed. It
// will display the elapsed time and update the previous time variable
// so that a number of blocks of code can be timed.
#ifdef PROFILING
    dwCurTime = GetCurrentTime();
    wsprintf((LPSTR)szProfBuf, (LPSTR)"%lu\n\r\0",
            dwCurTime - dwPrevTime);
    OutputDebugString((LPSTR)szProfBuf);
    dwPrevTime = dwCurTime;
#endif
```

The sample code listed above displays the elapsed time on a secondary debugging monitor. If required, this code can be modified to write the data to a file for analysis instead of displaying the data on a monitor.

Additional reference words: 3.00 MICS3 T13
KBCategory:
KBSubcategory: KrDebugMisc

INF: Tracking Unrecoverable Application Errors Without CVW
Article ID: Q68825

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

If an application causes an unrecoverable application error (UAE), and this behavior is not duplicated when the application runs under CodeView for Windows (CVW), it can be very difficult to isolate the problem.

If a debugging terminal is available, a call to `OutputDebugString()` will put a message on the terminal. However, if no additional hardware is in place, the application can leave data regarding its path of execution by calling `WriteProfileString()` to modify the WIN.INI file.

More Information:

Consider the following code fragment:

```
case WM_MOUSEMOVE:
    WriteProfileString ("TestApp", "Debug", "Entering WM_MOUSEMOVE");
    MousePos[iGlobalIndex] = lParam;
    WriteProfileString ("TestApp", "Debug", NULL);
    break;
```

Assume that it is unclear whether the UAE happens in this code (for example, if `iGlobalIndex` takes on an invalid value). This has not occurred when running in the CVW debugger, or it takes so long to reproduce under CVW that the debugger is not useful.

The next time the UAE occurs, if the [TestApp] section of the WIN.INI file has the following line

```
Debug=Entering WM_MOUSEMOVE
```

the UAE occurred in the bracketed code. Otherwise, the line would be deleted from WIN.INI. This evidence would be helpful in tracking and correcting the bug.

In this particular case, the performance hit can be quite severe because the `WM_MOUSEMOVE` message is sent quite often and updating WIN.INI requires a write to the disk. However, by testing an appropriate, but less common, message (such as `WM_LBUTTONDOWN*`) first, the worst case slowdown shown in the example above can be avoided.

Additional reference words: 3.00 MICS3 R3.9

KBCategory:

KBSubcategory: KrDebugGpfaulsts

FIX: One Cause of FatalExit 0x0403

Article ID: Q69805

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.0 and 3.1
-

Summary:

PROBLEM ID: WIN9103026

SYMPTOMS

When running an application under the debugging version of Windows version 3.0, Windows reports fatal exit 0x0403 "invalid ordinal reference."

CAUSE

An exported function in a Windows dynamic-link library (DLL) was declared with the RESIDENTNAME attribute in the DEF file associated with the DLL.

RESOLUTION

Microsoft has confirmed this to be a problem in Windows version 3.0. Either of the following will resolve this problem:

- Do not associate the RESIDENTNAME attribute with any exported function other than the Windows exit procedure (WEP) of the DLL. The WEP MUST be declared with the RESIDENTNAME attribute.

-or-

- The application can declare links to the functions of the DLL by using the IMPORTS section of its module definition file. When this is done, it is not necessary to use the IMPLIB utility.

For example, a DLL's module definition file, DLL.DEF, contains the following text:

```
EXPORTS
    zippo    @2      RESIDENTNAME
    harpo    @3      RESIDENTNAME
```

An application can avoid this problem by using the following text in its module definition file, APP.DEF:

```
IMPORTS
    zippo    = mydll.2
    harpo    = mydll.3
```

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00 SR# G910108-2

KBCategory:

KBSubcategory: KrDebugFatlexit

FIX: One Cause of FatalExit in Debug Enhanced Mode

Article ID: Q70800

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9103008

SYMPTOMS

Running an application under the Microsoft Windows debugging kernel causes a FatalExit message "free memory overwrite at y:x" to appear on the debugging monitor or terminal.

CAUSE

For performance reasons, when an application releases a memory page that was previously on the kernel free page list, Windows does not write the kernel's memory test pattern back into the freed memory. The kernel interprets this as a problem with the application.

RESOLUTION

Microsoft has confirmed this to be a problem in Windows version 3.0. Two ways to avoid this problem are as follows:

- During the debugging phase of application development, turn off paging by setting Paging=0 in the [386enh] section of the SYSTEM.INI file. Changing this setting negatively impacts system performance.
- Modify the [kernel] section of the WIN.INI file to set EnableFreeChecking=0.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrDebugFatlexit

FIX: UAE at Application Load Time Caused by Preload Area Size
Article ID: Q70805

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9103013

SYMPTOMS

When an application is in the process of loading, it experiences an unrecoverable application error (UAE).

CAUSE

The size of the preload area, as reported by the Microsoft Windows Resource Compiler, is an exact multiple of 64K (10000h).

RESOLUTION

Microsoft has confirmed this to be a problem in Microsoft Windows version 3.0. To avoid this problem, change the size of the preload area by modifying the application's module definition (DEF) file or resource (RC) file to change the number of segments that are marked PRELOAD.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrDebugMisc

INF: Redirecting Debugging Information Under Windows 3.0, 3.1
Article ID: Q86263

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

Each new application for the Microsoft Windows environment should be tested under the Windows debugging kernel. When an application performs an illegal or potentially harmful operation, the debugging kernel traps the error and provides a descriptive message about the source of the problem. By default, the debugging kernel sends its messages to the AUX device (which maps to the COM1 port). This article describes how to redirect output from the debugging kernel under Windows versions 3.0 and 3.1.

More Information:

As noted above, in the Windows 3.0 and 3.1 environments, the debugging kernel sends its data to the AUX device, which (in general) maps to the COM1 port. One method to display this debugging information is to connect a serial communications terminal or another computer running a terminal emulation program to the COM1 port. Before running Windows, set the port's parameters appropriately through the MS-DOS MODE command. Placing the MODE command in the AUTOEXEC.BAT file automatically sets the port's parameters.

When the debugging kernel starts, if no device is connected to COM1, the kernel displays a "Cannot write to device AUX" message box. If the COM1 port is dedicated to another use and cannot be connected to a serial terminal, you can configure the debug kernel to send its messages elsewhere. The remainder of this article explains the procedures required.

Windows 3.1

In the Windows 3.1 environment, the debugging terminal is not required. The Windows 3.1 debugging kernel provides two methods to redirect debugging information:

1. Redirect the debugging information from COM1 into a file by specifying the following in the [Debug] section of the SYSTEM.INI file:

```
OutputTo = <filename>
```

To disable sending debug messages to AUX, specify the following in the SYSTEM.INI file:

OutputTo = NUL

2. Version 3.1 of the Microsoft Windows Software Development Kit (SDK) includes the DBWIN sample program in the advanced samples directory (by default, C:\WINDEV\SAMPLES). DBWIN provides a good interface and some useful features to debug an application in the Windows environment. DBWIN can disable sending debugging messages to AUX or redirect the debugging messages to any of the following:

- The COM1 port
- The COM2 port
- A window on the primary display
- A secondary monochrome monitor

These redirection options are listed on the Options menu. When using DBWIN, choose Settings from the Options menu and verify that the Break On Traces option is not selected. For more information on DBWIN, see the DBAPI.TXT and DBWIN.TXT files in the DBWIN directory, Appendix C of the "Microsoft Windows Software Development Kit: Programming Tools" version 3.1 manual, and the online help files.

Windows 3.0

Windows 3.0 does not provide a built-in method to redirect debugging output; external measures are required.

This article outlines two methods to redirect output under Windows 3.0. The first is through the WINAUX.SYS device driver that redirects debugging output into a window on the main display, similar to the DBWIN application discussed above. The second is through the OX.SYS device driver that redirects debugging output to a secondary monochrome adapter connected to the system.

Because systems that use an 8514 and VGA display combination cannot also use a secondary monochrome monitor, the WINAUX.SYS device driver is the method of choice. WINAUX can be found in the Software/Data Library by searching on the word WINAUX, the Q number of this article, or S13525. WINAUX was archived using the PKware file-compression utility.

To install WINAUX.SYS, place the following line in the CONFIG.SYS file:

```
DEVICE=WINAUX.SYS
```

Another method to redirect debugging output under Windows 3.0 is to use the OX.SYS device driver that redirects output for the AUX device to a monochrome video adapter. Many development systems have a secondary monochrome display to use with CodeView for Windows (CVW). OX.SYS sends the debug messages to the monochrome display.

The OX.SYS file and its source code is available in the Software/Data Library. If necessary, you can modify the OX source code to direct debugging output to another device such as LPT1, COM2, and so on. OX can be found in the Software/Data Library by searching on the word OX,

the Q number of this article, or S12005. OX was archived using the PKware file-compression utility.

To install OX.SYS, add the following line to the CONFIG.SYS file:

```
DEVICE=OX.SYS
```

Under Windows 3.0, OX.SYS is limited because it provides an input-only or output-only device. Therefore, when the Windows debugging kernel provides output for a FatalExit message followed by the "Abort, Break, Ignore?" prompt, OX.SYS cannot obtain the user's response. Third-party developers have developed bi-directional device drivers to address this limitation and have placed the drivers into the public domain. Two examples are WINRIP.SYS and MONO-DRV.SYS.

Additional reference words: 3.00 3.10 softlib OX.ZIP WINAUX.ZIP

KBCategory:

KBSubcategory: KrDebugDebugver

PRB: Linker Warning L4000

Article ID: Q72384

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOM

During the linking of an application for Windows version 3.00, a warning L4000 is displayed by the Microsoft linker.

CAUSE

This warning is caused by the /WARNFIXUP linker switch. The warning is given when segments in the executable file are moved by the linker.

RESOLUTION

There are two ways to address this warning message:

1. Add the /NOPACKCODE option to the linker command line.
2. Modify the DEF file to include a SEGMENTS statement.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrDebugMisc

FIX: Bad Extended Error Information After Critical Error
Article ID: Q72495

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9012019

SYMPTOMS

After an MS-DOS critical error has occurred in Windows (for example, the system error "Cannot read from drive A:"), calls to return extended error information, such as MS-DOS Interrupt 21H function 59H or the C run-time function gettexterr, return the value 53H (Interrupt 24H failure) rather than the error code for the critical error.

CAUSE

Windows does not save the original error code to return to the application.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00 SetErrorMode MICS3 R3.5

KBCategory:

KBSubcategory: KrDebugMisc

PRB: Fatal Exit 0x00FF: MakeProcInstance for Current Instance
Article ID: Q74363

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

An application running under the Microsoft Windows graphical environment receives FatalExit error 0x00FF, "MakeProcInstance only for Current Instance."

CAUSE

A function in a dynamic-link library (DLL) calls the MakeProcInstance function using an application's instance handle. The FatalExit occurs because the current value of the DS register does not match the instance handle passed to MakeProcInstance.

RESOLUTION

If a function in a DLL calls MakeProcInstance on behalf of an application, the function must be exported as a NODATA function in the module definition (DEF) file for the DLL. This causes the function to use the same data segment as the calling application.

More Information:

When an application is executing, the value of the DS register is equal to the application's data segment, which is identified by the application's instance handle.

By default, when an application calls an exported DLL function, the value of the DS register is set to the data segment of the DLL. However, if the DLL function is exported with the NODATA option, the value of the DS register does not change when the function is called. In this case, the DLL can safely call the MakeProcInstance function using the calling application's instance handle because the current value of the DS register is the same as the instance handle passed to MakeProcInstance.

Additional reference words: 3.00 RIP FF

KBCategory:

KBSubcategory: KrDebugFatlexit

PRB: KRNL386: Unable to Enter Protected Mode

Article ID: Q105207

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
-

SYMPTOMS

=====

Windows 3.1 fails to boot and returns to MS-DOS with a "KRNL386: Unable to enter Protected Mode" error message. This usually occurs after switching to the debug version of Windows.

CAUSE

=====

Windows 3.1 was started with old files left over from an upgraded installation of Windows 3.0 with the Windows 3.0 Software Development Kit (SDK).

RESOLUTION

=====

Replace the old files with the correct versions. The files of importance are KRNL386.*, KRNL286.*, GDI.*, and USER.*. The old files are from 1990 and the new files are from 1992. Sort by date, and replace these old files with the correct versions.

MORE INFORMATION

=====

It's generally a good idea to start with a fresh installation of Windows in a clean directory when this problem occurs. This ensures that other obsolete files are cleared from the upgraded 3.0 installation.

Additional reference words: 3.10 3.00 n2d debug version d2n

KBCategory:

KBSubcategory: KrDebugDebugver

PRB: Windows FatalExit 0x0280 Error Caused by FAR WinMain
Article ID: Q41451

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

When an application is launched, a FatalExit 0x0280 (ERR_GMEMHANDLE, invalid global handle) error occurs.

CAUSE

The application declares the WinMain function as a FAR function. The design of the Microsoft Windows kernel assumes that an application's entry point is a near function rather than a far function.

RESOLUTION

Remove the FAR keyword from the declaration of the WinMain function.

Additional reference words: 1.x 2.03 2.10 3.00 2.x

KBCategory:

KBSubcategory: KrDebugFatlexit

INF: Stack Traces Under Windows 3.1 SDK Debugging Kernel
Article ID: Q89331

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

When a FatalExit occurs, the Microsoft Windows operating system version 3.1 Software Development Kit (SDK) debugging kernel does not display a stack trace on the debugging monitor unless the user presses the ENTER or SPACEBAR key immediately after the kernel displays the "Abort, Break, Ignore" message. This behavior is different from that found in the debugging kernel for Windows version 3.0 or prerelease versions of Windows version 3.1.

The rationale behind changing this behavior was to simplify the interface to the debugging kernel and to speed its execution. When a FatalExit message occurs, a stack trace is available if desired. However, the kernel does not take time to create and display unwanted stack traces.

More Information:

In its default configuration, the Windows debugging kernel displays messages on a serial terminal connected to the COM1 port. The kernel produces four levels of messages: Trace, Warning, Error, and FatalExit. Appendix C of the SDK "Programming Tools" manual and the SDK Help system documents the Windows debugging kernel.

When the debugging kernel displays the "Abort, Retry, Ignore" message for a FatalExit it does not display a stack trace immediately. Instead, the kernel enters a loop, waiting for the user to respond. If the user presses the SPACEBAR or ENTER key before the loop times out, the kernel displays the stack trace. To continue execution after the stack trace, press the I key to ignore the FatalExit. The other options are to press the A key to abort execution or the B key to break into the debugger.

The Windows 3.1 SDK includes an advanced sample application called DBWIN that provides a good user interface and some useful features to assist in debugging a Windows application with the debugging kernel. If the advanced samples are installed into the default directory, the DBWIN source code is in the C:\WINDEV\SAMPLES\DBWIN directory.

DBWIN can redirect debugging messages into a window on the main display or to a secondary monochrome monitor. However, when DBWIN redirects messages in this manner, the debugging kernel ignores FatalExit messages (irrespective of the debug settings). In other words, no stack traces are available when DBWIN redirects debug messages to a window or a secondary monochrome monitor. However, stack traces are available when DBWIN redirects debugging information

to COM1 or COM2 as outlined above for a debugging terminal.

DBWIN ignores FatalExit messages because the system runs much faster when it displays debugging messages in a window rather than on a serial terminal. However, because a stack trace provides very useful information to assist in debugging an application, this default behavior might not be considered very useful.

The text below provides the modification to the DBWIN source code required to provide stack traces in a window or on a secondary monochrome monitor. The modified version of DBWIN produces a stack trace for every FatalExit message displayed by the debugging kernel, similar to the behavior of the Windows 3.0 debugging kernel. While the system might run slowly with the modified DBWIN, the additional debugging information might make the change worthwhile. The modified version of DBWIN is available in the NUDBWIN file in the Software/Data Library. NUDBWIN can be found in the Software/Data Library by searching on the keyword NUDBWIN, the Q number of this article, or S13596. NUDBWIN was archived using the PKware file-compression utility.

The only modifications required are to the NotifyCallback function in the DBWINDLL.C source file. Add the text in the lines that begin with NEW to the file, as follows:

```
BOOL CALLBACK _export _loads NotifyCallback(WORD id, DWORD dwData)
{
    BOOL fHandled;
    .
    .
    NEW // By default, produce stack trace at every FatalExit
    NEW static BOOL fStackTrace = TRUE;

    // If we're not outputting anything,
    // just return FALSE to chain to next handler.
    if (modeOutput == OMD_NONE)
        return FALSE;

    .
    .
    .

    case NFY_INCHAR:
        switch (modeOutput)
        {
            case OMD_COM1:
            case OMD_COM2:
                fHandled = (BOOL)ComIn();
                break;

            default:
                NEW if (fStackTrace)
                NEW     fHandled = (BOOL)' '; // Return a SPACEBAR press
                NEW     // to produce stack trace
                NEW else
                NEW     fHandled = (BOOL)'i'; // Return an I key press to
```

```
NEW                                     // ignore the FatalExit
NEW
NEW     // Do not produce the stack trace a second time at the
NEW     // "Abort, Break, Ignore" message. Ignore FatalExit this time
NEW     fStackTrace = !fStackTrace;
      }
      break;

      .
      .
      .
```

Additional reference words: 3.10 RIP

KBCategory:

KBSubcategory: KrDebugstacktr

PRB: One Cause of Fatal Exit 0x0140

Article ID: Q75359

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

The debugging version of Windows 3.0 reports FatalExit 0x0140.

CAUSE

A module with no heap (such as a no-data dynamic link library) uses LocalAlloc to allocate local memory. Even if the application does not call LocalAlloc directly, application startup code in the C run-time libraries allocates memory to store the command-line arguments.

RESOLUTION

Make sure that the proper library is specified on the LINK command line.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrDebugFatlexit

PRB: Strange UAE in Windows 3.00

Article ID: Q75502

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

The application is terminated without problems; however, running Microsoft Excel or Word for Windows results in either an unrecoverable application error (UAE) or the computer hangs. Analysis of Dr. Watson logs indicate a general protection fault at:

SetHandleCount + 58

The problem occurs with Windows version 3.0 in standard mode with large applications, or multiple executables and DLLS (dynamic-linked libraries) with at least one of the DLLs being a large-model DLL.

CAUSE

Windows version 3.0 does a scan of the LDT (local descriptor table) and selects an invalid selector.

RESOLUTION

The problem disappears in Windows version 3.0 if the application performs a GlobalCompact(-1) call.

Microsoft has confirmed that this problem has been resolved in Windows version 3.1.

Additional reference words: 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrDebugGpfaults

PRB: One Cause of Fatal Exit 0x001A

Article ID: Q75737

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

Closing an application or dynamic-link library (DLL) causes Windows to issue the undocumented fatal exit 0x001A.

CAUSE

The application or DLL has registered a window class with the CS_GLOBALCLASS style. The fatal exit is issued when Windows terminates the application or DLL that registered the class, when a window of that class is still open.

RESOLUTION

Ensure that all windows of any classes registered by an application or DLL are closed before the application or DLL is terminated.

More Information:

Applications and DLLs can register window classes that are visible to all applications by using the class style CS_GLOBALCLASS. This style is most commonly used in custom-control DLLs that are meant to be shared by multiple applications.

When a task (that is, an application) terminates, all classes registered by that application are unregistered. In addition, DLLs that were implicitly loaded by the terminating application are freed if only the terminating application is using the DLL. In a similar fashion, during the unload sequence of a DLL, any classes that it registered are also unregistered.

Windows keeps an internal count of windows created with a specific class. If this count is not zero when a class is unregistered, Windows reports fatal exit 0x001A.

Additional reference words: 3.00 RIP 1A MICS3 R3.14

KBCategory:

KBSubcategory: KrDebugFatlexit

INF: Checking for Invalid Global or Local Handles

Article ID: Q77472

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

ISHANDLE is a file in the Software/Data Library that contains the source code to an application for the Microsoft Windows environment called HANDLE.EXE and to a dynamic-link library (DLL) called ISHANDLE.DLL. The DLL exports two functions, IsGlobalHandle and IsLocalHandle, that validate global and local handles, respectively. The HANDLE application links into the DLL and tests the two functions. ISHANDLE calls functions exported by the Tool Helper library and is compatible with Windows 3.0 and 3.1.

ISHANDLE can be found in the Software/Data Library by searching on the word ISHANDLE, the Q number of this article, or S13210. ISHANDLE was archived using the PKware file-compression utility.

More Information:

The IsLocalHandle and IsGlobalHandle functions use functions exported by the Tool Helper library to validate the given local or global handle. ISHANDLE uses the LocalFirst and LocalNext functions to walk the local heap until it finds the desired handle or reaches the end of the heap. Similarly, ISHANDLE uses the GlobalFirst and GlobalNext functions to walk the global heap.

For example, the following demonstrates how IsLocalHandle walks the local heap and validates a specified local handle, hLocalHandle:

```
// Declare local variable
LOCALENTRY leTemp;

// Allocate a buffer to do the local heap walk
bFound = FALSE;
leTemp.dwSize = sizeof(LOCALENTRY);

// Loop through the local heap until hLocalHandle is found
if (LocalFirst(&leTemp, wHeap))
{
    do
    {
        if (leTemp.hHandle == hLocalHandle)
        {
            bFound = TRUE;
            break;
        }
    } while (LocalNext(&leTemp));
}
```

```
    }  
    if (bFound)  
        return hLocalHandle;  
    else  
        return NULL;
```

While the code above illustrates simply validating a handle, an application can extend the process to gather additional information about each memory block from the LOCALENTRY or GLOBALENTRY data structures. For example, an application can build a dynamic list of information about the heaps.

Additional reference words: 3.00 3.10 softlib ISHANDLE.ZIP

KBCategory:

KBSubcategory: KrDebugMisc

INF: GetDriveType DRIVE_REMOVEABLE Documentation Error
Article ID: Q66394

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

On Page 4-176 of the "Microsoft Windows Software Development Kit Reference Volume 1" version 3.0, the spelling of the "DRIVE_REMOVEABLE" return value from the GetDriveType function is incorrect. It should be spelled DRIVE_REMOVABLE (without the "E" between "MOV" and "ABLE") to match the spelling in WINDOWS.H.

Microsoft has confirmed that this documentation error has been corrected on page 368 of the "Microsoft Windows Software Development Kit Programmer's Reference, Volume 2: Functions" for version 3.1.

Using the documented value name in C source code will cause the Microsoft C Compiler to produce the following error message during compilation:

```
error C2065: 'DRIVE_REMOVEABLE' : undefined
```

Additional reference words: 3 3.0 3.00

KBCategory:

KBSubcategory: KrDskDrivetype

INF: Writing Volume Labels to Floppy and Hard Disks

Article ID: Q71498

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

To write a volume label on a floppy or a hard disk, a Windows application can use the C run-time function `_dos_create()`. However, if the disk already has a volume label, `_dos_create()` fails unless the old label is first deleted.

SETVOL is a sample in the Software/Data Library that demonstrates writing a volume label to the disk in drive A. SETVOL makes use of the Extended File Control Block Structure and uses some in-line assembly to check whether a volume label exists. If so, SETVOL deletes the label.

SETVOL can be found in the Software/Data Library by searching on the word SETVOL, the Q number of this article, or S13006. SETVOL was archived using the PKware file-compression utility.

Additional reference words: 3 3.0 3.00

KBCategory:

KBSubcategory: KrDskVollabels

PRB: Windows Applications Cannot Share File Handles

Article ID: Q22379

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

One application initializes a file, and receives the handle of x '0005'. The application then takes the handle and places it into global memory. The SetClipboardData function is then called using a format returned by the RegisterClipboard function. A second application then opens the Clipboard and retrieves the handle correctly; however, when the application tries to write to the handle, MS-DOS returns AX=0000, indicating that it did not write any information.

RESOLUTION

Applications cannot share file handles. This is a feature of the MS-DOS filing system.

The file handle table is part of each application's environment; the file handle itself is an offset into this table. Although an offset of 5 might be valid for two applications, the file information at that offset in the respective file handle tables will be very different.

Passing the filename to the second application is recommended.

Additional reference words: TAR60849 2.x 2.00 2.03 2.10 2.x 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrFileioFileshare

FIX: sopen() Fails When Called from a Windows DLL

Article ID: Q72458

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9102002

SYMPTOMS

When the sopen function is called from a Windows dynamic-link library (DLL), the file sharing flags are ignored.

CAUSE

The sopen function refers to a C run-time library variable, `_osmajor`, to determine whether file sharing is supported. This variable is not initialized in the DLL version of the Microsoft C version 6.0 run-time libraries.

RESOLUTION

Microsoft has confirmed this to be a problem in the Microsoft C run-time libraries for DLLs provided with the Windows Software Development Kit version 3.0. To avoid this problem, declare an unsigned char variable, `_osmajor`, in a module that uses the sopen function and assign the variable a value of 3 or higher.

This problem was corrected in the Microsoft C/C++ Optimizing Compiler version 7.0.

Additional reference words: 3.00 6.00 7.00

KBCategory:

KBSubcategory: KrFileioFileshare

INF: Limits on the Number of Open Files

Article ID: Q81577

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Because an application developed for the Windows graphical environment runs only within the MS-DOS environment, the number of files that the application can open is subject to two distinct limits imposed by MS-DOS. This article discusses these limits.

More Information:

The first limit on the number of open files is imposed by a table of SFT data structures (System File Tables) within MS-DOS. The initial number of SFTs is specified in the FILES= line of the CONFIG.SYS file. MS-DOS version 5.0 does not provide a function to change the number of SFTs in the table. However, some applications, including Windows, contain the information necessary to change the size of the table.

The SFT table is global, that is, it is shared by all applications or tasks that are active in the system. (Each task is represented by a PSP or program segment prefix.) The exception to the global nature of the SFT table, and the associated limit on the number of open files, is introduced by Windows. Different groupings of applications, called virtual machines (or VMs), have a "per VM" address space. This allows a virtual machine to have a local portion of the SFT table, which exists only in that VM. Because all graphical applications in the Windows environment run in a single VM, only the MS-DOS (not graphical) applications run in separate VMs.

MS-DOS does not support more than 255 SFTs. Therefore, it does not support more than 255 open files at any time in any specific VM. The SHARE utility, which extends MS-DOS core functionality, does not support the ability to increase the size of the SFT table on a "per VM" basis under Windows, because SHARE must, at any time, in any VM, be able to enumerate all open SFTs in the entire system.

The second limit on the number of open files involves a "per application" table, called the JFN table, which is stored in each task's PSP. By default, the JFN in the PSP has room for 20 entries, which limits each application to 20 open files. An application running on MS-DOS version 3.3 and subsequent versions can change the size of the JFN by calling MS-DOS INT 21h function 67h. This call allocates a new JFN table and modifies values in the PSP to indicate the larger size of the JFN table. This allows the application corresponding to a particular PSP to open more than 20 files, provided that the global SFT table, shared by all applications, has available SFTs.

The two limits work together as follows. First, to open a file, the application must have a place available in its JFN table. Second,

MS-DOS must have an available SFT in its internal SFT table. Note that if either of these requirements is not met, the OPEN (or CREATE) INT 21h call will fail with error 4.

Additional reference words: 3.00 3.30 5.00

KBCategory:

KBSubcategory: KrFileioSethandle

INF: File Manager's Mechanism for Sensing File System Changes
Article ID: Q67725

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The Windows version 3.0 File Manager automatically updates its directory information any time a Windows or a non-Windows application creates, renames, or deletes a file.

This information is not available for use by application developers, and there are no future plans to make this information public.

Additional reference words: 3 3.0 3.00

KBCategory:

KBSubcategory: KrFileioMisc

INF: Windows OpenFile Function vs. C Run-Time

Article ID: Q11988

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

In Windows, the OpenFile function creates an MS-DOS file handle through which an application can access Windows-specific files. OpenFile initially opens the file in binary raw mode by performing an MS-DOS Interrupt 21h Function 3Dh. If the lpFileName parameter specifies only a filename and an extension, OpenFile searches for a matching file in the following directories:

- The current directory.
- The Windows directory. The GetWindowsDirectory function returns the path to this directory.
- The Windows system directory . The GetSystemDirectory function returns the path to this directory.
- The directories listed in the PATH environment variable.

The open, fopen, and sopen functions provided by the Microsoft C run-time libraries and the _lopen function provided by Windows can be used to access any file. The open functions do not necessarily open a file in binary raw mode; the application is required to set the binary attribute explicitly. The OpenFile function automatically performs this step.

If the filename parameter specifies only a filename and extension, the open functions search for a matching file only in the current directory.

An application should use the OpenFile function any time an MS-DOS file handle is required.

Additional reference words: 2.x TAR56861

KBCategory:

KBSubcategory: KrFileioFileapi

BUG: sopen() Fails When Called From a Windows DLL

Article ID: Q68942

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The sopen() function provided by Microsoft C versions 6.0 and 6.0a is listed as being compatible with Windows dynamic-link libraries (DLLs). However, when sopen is called from a DLL, it ignores the file-sharing flags and fails to perform as documented.

More Information:

When it is called from a DLL, sopen() refers to the C library variable `_osmajor` to determine if the MS-DOS version is 3.0 or later. These versions of DOS support file sharing. When sopen makes this check, `_osmajor` contains the value 0 (zero), and sopen ignores the file-share flags. The `_osmajor` variable is 0 because no C run-time library initialization is performed for Windows DLLs.

This problem can be avoided by performing the following two steps before using sopen():

1. Declare an unsigned char `_osmajor`.
2. Assign `_osmajor` the value 3 or higher.

Microsoft has confirmed this to be a problem in Windows versions 2.03 and 2.1 and in the DLL libraries for Windows version 3.0. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 2.03 2.10 3.00 6.00 6.00a

KBCategory:

KBSubcategory: KrFileioFileshare

INF: Updating Cached Private Profiles (.INI Files)

Article ID: Q68827

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

Under Windows version 3.1, the first time a private profile (.INI file) is accessed, the system will call the `GetFileTime` function and store this value. The `WriteProfileString` function will then call the `GetFileTime` function and compare the return value to the stored value. If the two values match, the file is considered valid for two seconds. The function makes the changes and writes the new contents to disk. If the two values do not match, the profile is reread into a buffer and the change is made. The same principle holds true for reading values from a private profile.

The reasoning behind the two second limit is that most applications read private profiles in a burst, at application startup, and write in a burst, at application shutdown. The penalty of one read in a twenty read sequence is considered acceptable, given the benefits.

In Windows version 3.0, an application that has a private profile will not respond to changes made to that private profile by a text editor. When a text editor updates a private profile, the file on disk is modified. However, the `GetPrivateProfileString` and `GetPrivateProfileInt` functions do not read from the disk file, instead the functions read from a copy of the file in a cache. The `WritePrivateProfileString` and `WritePrivateProfileInt` functions will update the appropriate sections in both the cached file and the disk file, however, the functions will not reload the disk file into the cache unless the entire cache is invalidated. The information included below discusses how to force a private profile to be recached from a disk file.

More Information:

Windows caches .INI files to reduce access time. This design allows the file to remain in memory until a different .INI file is loaded or until an application forces recaching of the file.

To force an .INI file to be recached, make the following call (where `<fname.ini>` is the name of the application's private profile):

```
WritePrivateProfileString(NULL, NULL, NULL, <fname.ini>)
```

This call will force the entire .INI file that is in the cache to be invalidated. The next call to either the `GetPrivateProfileString` or `GetPrivateProfileInt` functions will cause the disk file to be recached.

While .INI files are cached to optimize access time, the following are examples of how and when an .INI file could be recached.

1. The application could update the cache from disk each time the application requires information from the profile. Calling the WritePrivateProfileString function as outlined above would clear the cache.

Note: Because the file is recached with every access, the benefit of the cache is lost with this method.

2. Create a separate program or function that the user would invoke to explicitly invalidate the cache. The following is some code for that purpose that could be placed into the GENERIC sample application supplied with the Windows Software Development Kit (SDK):

```
BOOL InitInstance(HANDLE hInstance, int nCmdShow)
{
    LPSTR lpApplicationName, lpKeyName, lpDefault, lpReturnedString;
    int    nSize;

    /* initialize variables */
    ...

    WritePrivateProfileString(NULL, NULL, NULL, "MY.INI");
    GetPrivateProfileString(lpApplicationName, lpKeyName,
        lpDefault, lpReturnedString, nSize, "MY1.INI");
    MessageBox(NULL, "Cache Refreshed", szApp,
        MB_ICONINFORMATION | MB_OK);
    return TRUE;
}
```

Using a program or function like this will cause the .INI file to be recached only when it is changed by an editor, therefore the benefit of the cache is retained. However, it is necessary for the user to call another application or function after the profile is changed with an editor.

3. If neither of these techniques is suitable, the application could check the time and date stamp on the .INI file before each access to see if cache invalidation is necessary. This option provides the benefits of the cache without requiring the user to call another program. The overhead required to read the time and date stamp is minimal compared to recaching the file with every call to either the GetPrivateProfileString or GetPrivateProfileInt functions.

Additional reference words: 3.00 3.10 3.x SR# G910109-169

KBCategory:

KBSubcategory: KrFileioMisc

INF: Handling Critical Errors in a Windows Application

Article ID: Q69027

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

The Windows SetErrorMode() function controls whether Windows handles MS-DOS Function 24H errors or allows the calling application to handle them. Listed below is an example of an MS-DOS Function 24H error and Windows's method for handling the error: if an application attempts to access drive A and there is no disk in that drive, Windows displays the System Error message box "Cannot read from drive A: (Cancel) (Retry)."

Calling SetErrorMode() allows the application to handle these messages rather than defaulting to the System Error message box.

The following code fragment demonstrates this process:

```
SetErrorMode(1);    // Allow application to handle system error
...
/* If error occurs, handle it appropriately. */
...
SetErrorMode(0);   // Windows will display the standard
                  // INT 24H error message box for any other
                  // System Errors.
```

For more information on a problem in Windows version 3.0 with extended error handling, query on the following keyword:

WIN9012019

Additional reference words: 3 3.0 3.00

KBCategory:

KBSubcategory: KrFileioMisc

INF: Opening Files, Compatibility Mode and Windows

Article ID: Q74445

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Opening a file in compatibility mode is a very unfriendly action in a multitasking environment such as Windows. There is never any need to do this; compatibility mode provides support for old MS-DOS (non-Windows) applications however it does not provide additional functionality.

Use the following four guidelines for opening files:

1. Do not use the `_lcreate` function, it opens files in compatibility mode. If it is necessary to create a file, immediately close the file and then reopen it with the `_lopen` function.
2. Do not use the `OF_SHARE_COMPAT` option with the `_lopen` or `OpenFile` functions. Instead, use one of the other `OF_SHARE` defines. If no `OF_SHARE` value is specified, the file is opened in compatibility mode.
3. When creating a file using MS-DOS interrupts 3Ch or 5Bh, after creating the file, close it and then open it again. The create leaves it in compatibility mode.
4. When opening a file (MS-DOS interrupts 3Dh or 6Ch), make sure that the file is NOT opened in compatibility mode.

When specifying the open mode and share, do not request more access than required. If a file will only be read, open it in read-only mode **EVEN IF EXCLUSIVE ACCESS IS REQUESTED**. Do not lock out other access unnecessarily. If an application will only read a file, allow other applications to read the file as well.

Finally, be open to alternatives when opening a file. If a file is being opened to display its contents and an open `READ-ONLY, DENY-WRITE` fails, try an open `READ-ONLY, DENY-NONE`.

Additional reference words: 3.00 create open READ-ONLY share

KBCategory:

KBSubcategory: KrFileioFileshare

INF: Using OpenFile with Sharing and Inheritance Bits

Article ID: Q43397

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The following information describes the file-access, file-sharing, and inheritance codes used with MS-DOS Interrupt 21h Function 3Dh that are compatible with the Windows OpenFile function.

A call to the OpenFile function that specifies OF_CREATE as the value for the wStyle parameter is translated into a call to the MS-DOS Create File with Handle function (Interrupt 21h Function 3Ch). Because Windows does not pass any style bits (other than OF_CREATE) to MS-DOS, the file is always created with normal attributes.

A call to the OpenFile function that does not specify OF_CREATE is translated into a call to the MS-DOS Open File with Handle function (Interrupt 21h Function 3Dh). Windows places the value of the low-order byte of the wStyle parameter into the AL register to specify file-access and file-sharing codes.

Note: The sharing bits take effect only if the MS-DOS SHARE utility is running on a system.

Additional reference words: 2.03 2.10 3.00 3.10 2.x SR# G880926-3697

KBCategory:

KBSubcategory: KrFileioFileapi

INF: Application Dynamically Links to a DLL Using a Class
Article ID: Q85282

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

The traditional methods an application uses to dynamically link to a dynamic-link library (DLL) in the Windows environment, using the LoadLibrary and FreeLibrary functions, can be awkward. The application is required to call GetProcAddress for each DLL function the application links to, and the application is then required to store the returned address in an array. The source code of the application must also contain a prototype for each called function or the programmer must cast each function parameter to the required type.

The object-oriented techniques of Microsoft C/C++ version 7.0 can be applied to ease the process of dynamically linking with a DLL. An application can define a class that links to the DLL and contains pointers to each of the exported functions. Member functions of the class correspond to the exported functions in the DLL.

DYNDLL is a file in the Software/Data Library that demonstrates using a class, called CDynDLL, to dynamically link to a DLL. The CDynDLL constructor loads the library and retrieves pointers to each function exported by the DLL. The CDynDLL destructor frees the library. The member functions of the CDynDLL class correspond to the functions exported by the DLL.

DYNDLL can be found in the Software/Data Library by searching on the word DYNDLL, the Q number of this article, or S13450. DYNDLL was archived using the PKware file-compression utility.

Additional reference words: destructor constructor 3.10 softlib
DYNDLL.ZIP
KBCategory:
KBSubcategory: KrFileioFileshare

PRB: C Run-Time locking Function Causes Sharing Violations
Article ID: Q85284

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

The locking function in the Microsoft C run-time library may cause a sharing violation when it is called under the following conditions:

- The locking function is called from a dynamic-link library (DLL).
- The DLL is stored on a network drive.
- The locking function is called with the LK_UNLCK parameter to unlock a specified number of bytes.
- The unlocked bytes were not previously locked with a call to the locking function with the LK_LOCK parameter.

CAUSE

This error is the result of incorrect usage and should not be considered a problem with the Windows Software Development Kit (SDK) itself.

RESOLUTION

This problem occurs only with the libraries provided with the Windows SDK version 3.0. To work around this problem, rewrite the code that calls the locking function to unlock only bytes that have been previously locked.

More Information:

The following code demonstrates the problem described above:

```
/* Compile options needed: /Gsw /Zp
 *   Link options needed: /ALIGN:16
 */

#include <windows.h>
#include <io.h>
#include <fcntl.h>
#include <sys\locking.h>
#include <share.h>
#include <errno.h>

int _far _pascal LibMain(HANDLE h, WORD ds, WORD hs, LPSTR c)
    {return 1;}
void _far _pascal WEP(int nParam) {}
```

```
void _far _pascal testlock(void)
{
    int fd = sopen("m:\\network.fil", O_BINARY | O_RDWR, SH_DENYNO);
    if (fd >= 0)
    {
        int i;
        errno = 0;
        i = locking(fd, LK_UNLCK, 1L);
        close(fd);
    }
}
```

Additional reference words: 6.0 6.00 6.0a 6.00a 6.0ax 6.00ax 7.0 7.00

KBCategory:

KBSubcategory: KrFileioFileshare

INF: File Input/Output for Windows-Based Applications

Article ID: Q72237

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

An application written to work in a cooperative multitasking environment, such as that of Microsoft Windows, must be designed to work within specified constraints. This article discusses these constraints and the methods used to access files in Windows applications.

More Information:

The cardinal rule for file I/O in Windows is to not keep a file open for long periods of time. Specifically, a file should not be kept open during the processing of more than one message. An application should open a file, read or write data as appropriate, and close the file during the processing of one message.

An application can use either of two options to access a file that has been opened with the `OpenFile` function:

1. Use the C run-time library file I/O instructions.
2. Use the file I/O instructions provided by Windows.

The file handle returned from `OpenFile` can be used directly with the C run-time library "low-level" file I/O functions such as the `open`, `read`, `lseek`, `write`, `tell`, and `close` functions. One problem with these functions is that an application cannot use a FAR or HUGE pointer as a parameter to one of these functions unless the application is developed using the compact or large memory model. However, these two memory models are not recommended for applications for the Windows environment because the data segments must be fixed in memory. To work around the inability to use a FAR or HUGE pointer, the application must read data from the file into a local memory buffer and then copy the data to global memory.

The application can also use the C run-time library buffered I/O functions: `fopen`, `fread`, `fwrite`, and `fclose`. The `fdopen` function converts the file handle returned by `OpenFile` to a pointer to a FILE data structure. Buffered I/O is not very useful in Windows because an application should read and write large blocks of data to a file. Buffering is most helpful when small blocks of data are read and written.

The file I/O functions provided by Windows are: `_lopen`, `_lclose`, `_lcreate`, `_llseek`, `_lread`, and `_lwrite`. The "l" prefix indicates that each function accepts a FAR pointer to a buffer, which allows the application to transfer information in a file directly to global

memory and back. (Although these functions were part of each Windows release, they were first documented in Windows 3.0.)

Using the Windows file I/O functions is the easier method when data buffers are stored in global memory. Do not write assembly-language code to interface directly with the MS-DOS file I/O functions.

If an application uses stream I/O instead of the low-level I/O functions, the performance of the application may slow. This decrease in performance is caused by the buffering system used by the stream I/O functions. When an application calls the fopen function to open a file, functions in the C run-time library creates a file record that contains pointers into a stream buffer allocated from global memory. If the global heap does not have enough free memory to satisfy the buffer allocation, the I/O operation continues with a one character buffer.

There are two methods to address this situation. One method is to reduce the amount of data stored in the global heap. The other method is to develop the application using the compact or large memory model (neither of which is recommended in general for applications in the Windows environment). If the application is developed with an alternate memory model, it may be advantageous to specify the /Gt Microsoft C Compiler option switch to remove static data from DGROUP (the default data segment).

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrFileioFileapi

INF: LZEXPAND.DLL API Documentation

Article ID: Q75472

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

LZEXPAND.DLL is a dynamic-link library that provides functions to read, expand, and copy both regular files and files compressed by the Lempel-Ziv encoding program COMPRESS.EXE, provided with the Windows version 3.0 Software Development Kit (SDK). This article documents the services provided by LZEXPAND.DLL.

Note: This information applies ONLY to Windows version 3.0. The services provided in the next version of Windows will have a different interface.

More Information:

The LZInit, LZOpen, and LZClose functions perform necessary housekeeping (and open and close files). The LZRead and LZSeek functions can operate on both compressed and normal files. The LZCopy function copies files and decompresses them if necessary.

The LZSeek, LZRead, and LZClose functions can determine if the parameter passed in is a regular MS-DOS file handle or a compressed-file-information structure identifier, and will act appropriately. The LZOpenFile, LZSeek, LZRead, and LZClose functions can be called as replacements for the OpenFile, _llseek, _lread, and _lclose functions, respectively, without regard to the compression state of the files.

A detailed explanation of the purpose, syntax, and parameters of each LZEXPAND.DLL function is listed below:

LZCopy

Syntax: LONG LZCopy(doshSource, doshDest)

This function copies the source file to the destination file. If the source file is compressed, it is decompressed as it is copied.

Parameter	Type/Description
-----	-----
doshSource	int Specifies the MS-DOS file handle of the source file. The source file may be either compressed or uncompressed.
doshDest	int Specifies the MS-DOS file handle of the destination file. The destination file will be

an uncompressed file.

Return Value: The return value is the number of bytes written to the destination file, or it is one of the LZERROR codes listed below.

Code		Description
----		-----
LZERROR_BADINHANDLE	(-1)	Invalid input handle
LZERROR_BADOUTHANDLE	(-2)	Invalid output handle
LZERROR_READ	(-3)	Bad compressed file format
LZERROR_WRITE	(-4)	Out of space for output file
LZERROR_GLOBALLOC	(-5)	Insufficient memory for buffers
LZERROR_GLOBLOCK	(-6)	Bad global handle

Comment: The doshSource and doshDest parameters should be MS-DOS file handles returned either by the Windows OpenFile function, or by the MS-DOS open file functions (open, _dos_open, or _lopen). If the files are opened with the LZOpenFile function, the files will be copied properly; however, the LZOpenFile call will allocate an unused compressed-file-information structure for the compressed source file.

Using the LZCopy function is the fastest way to copy a file using LZEXPAND.

LZOpenFile

Syntax: int LZOpenFile(lpFileName, lpReOpenBuf, wStyle)

This function opens a file using the OpenFile function. If the file is opened in read-only mode and it is compressed, the LZOpenFile function also calls the LZInit function to allocate and initialize a compressed-file-information data structure.

Parameter	Type/Description
-----	-----
lpFileName	LPSTR Points to a null-terminated character string that names the file to open. The string must contain characters from the ANSI character set.
lpReOpenBuff	LPOFSTRUCT Points to the OFSTRUCT data structure to be used by the OpenFile function (for more information, see pages 4-322 to 4-325 of the "Microsoft Windows Software Development Kit Reference, volume 1").
wStyle	WORD Specifies action to be taken by the OpenFile function.

Return Value: If the file is opened in any mode other than read-only or if the file is not compressed, the LZOpenFile function returns the return value from the OpenFile function, an MS-DOS file handle. If the file was opened in read-only mode and it is compressed, the LZOpenFile function returns an identifier for the compressed-file-information data structure created by the LZInit function. If the file cannot be opened, -1 is returned. If the file can be opened, however the compressed-file-information data structure cannot be allocated, then the file is closed, and -1 is returned.

Comment: Files that are compressed with COMPRESS.EXE have a special header. The LZOpenFile function uses this header to determine if the file is compressed.

LZInit

Syntax: int LZInit(doshSource)

This function allocates and initializes a compressed-file-information data structure if it is passed a file handle for a compressed file.

Parameter	Type/Description
-----	-----
doshSource	int Specifies the MS-DOS file handle returned by the OpenFile, _lopen, or open functions. The file should have been opened read-only.

Return Value: If the file is compressed, an identifier for a compressed file information structure is returned. If the file is not compressed, the MS-DOS file handle is returned. If a read error occurs while initializing the data structure, or if there is insufficient heap space to allocate the structure, -1 is returned.

Comment: Storage for the compressed file information structure is allocated from the global heap.

LZSeek

Syntax: LONG LZSeek(hLZFile, lOffset, iOrigin)

This function repositions the pointer in a previously opened file.

Parameter	Type/Description
-----	-----
hLZFile	int Specifies the MS-DOS file handle or the identifier for compressed-file-information

data structure. It is best to open the file by calling the LZOpenFile function, which calls the LZInit function. However, the normal Windows OpenFile function can be used instead, followed by a call to the LZInit function to allocate and initialize the data structures used by the expansion algorithm.

lOffset LONG Specifies the number of bytes the pointer is to be moved.

iOrigin int Specifies the seek origin as used by the _llseek function.

Return Value: The return value is the new position in the file or -1 if the seek is unsuccessful.

Comment: If the file is not compressed, the LZSeek function calls the _llseek function with the MS-DOS file handle. If the file is compressed, the _llseek function is emulated on the expanded image of the file.

LZRead

Syntax: int LZRead(hLZFile, lpBuffer, wBytes)

If the file specified by hLZFile is not compressed, the LZRead function calls the _lread function to read the file. If the file is compressed, the file is read, decompressed, and copied into lpBuffer until either the EOF is reached or wBytes bytes have been written to lpBuffer.

Parameter -----	Type/Description -----
hLZFile	int Specifies the MS-DOS file handle or the identifier for compressed-file-information data structure for file to be read. It is best to open the file by calling the LZOpenFile function, which calls the LZInit function. However, the normal Windows OpenFile function can be used instead, followed by a call to the LZInit function to allocate and initialize the data structures used by the expansion algorithm.
lpBuffer	LPSTR Pointer to the buffer that is to receive the data read from the file.
wBytes	WORD Specifies number of bytes to be read from file.

Return Value: The return value specifies the number of bytes written to lpBuffer. It is less than wBytes only if the EOF has been reached. If the read fails, the return value is -1.

LZClose

Syntax: VOID LZClose(hLZFile)

This function closes the file specified by the hLZFile parameter. If the file is compressed, the global heap space occupied by the compressed file information structure is freed.

Parameter	Type/Description
-----	-----
hLZFile	int Specifies the MS-DOS file handle or the identifier for compressed-file-information data structure for the file to be closed.

Additional reference words: 3 3.0 3.00

KBCategory:

KBSubcategory: KrFileioExpcom

INF: No MS-DOS Extended Error Info for Windows File Functions
Article ID: Q86648

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

When an application uses one of the Microsoft Windows file functions (_lclose, _lcreat, _llseek, _lopen, _lread, or _lwrite) and an error occurs, the function returns the HFILE_ERROR error code. No additional information is available concerning the cause of the error. Specifically, no MS-DOS error return value is available for these functions.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrFileioFileapi

PRB: Creating File with Exclusive Access Allows Concurrent Use
Article ID: Q86723

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

SYMPTOMS

When an application calls the OpenFile function and specifies the OF_CREATE and OF_SHARE_EXCLUSIVE flags, the created file is not open for exclusive access. Another application can also open the file.

RESOLUTION

To create a file and open it for exclusive access, an application must create the file, close the file, and open it for exclusive access.

More Information:

The OpenFile function passes its parameters to MS-DOS Interrupt 21h. OpenFile and Interrupt 21 exhibit the same behavior in this regard.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrFileioFileshare

INF: Determining That SHARE Is Loaded Under Microsoft Windows
Article ID: Q72744

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

To determine whether the SHARE.EXE application is loaded, an application typically calls MS-DOS Interrupt 2Fh Function 1000h. However, this method always returns true in enhanced mode Microsoft Windows version 3.0 even if the SHARE is not loaded.

Windows returns true for Interrupt 2Fh Function 1000h to prevent SHARE from installing itself in a MS-DOS virtual machine (VM) under Windows. If SHARE installed a local copy in a VM, the system would become unstable and data corruption on the hard drive(s) might result.

More Information:

To determine under enhanced mode Windows whether SHARE is installed, call Interrupt 21h Function 5Ch to lock a region of a file. This function is available only when SHARE is installed, and unlike using the OpenFile function with sharing modes, the lock region function always fails with error 1 (invalid function) if SHARE is not loaded. Perform the following six steps to determine whether SHARE is loaded:

1. Create a temporary file using MS-DOS Interrupt 21h Function 5Ah.
2. Lock a region of the returned temporary file using MS-DOS Interrupt 21h Function 5Ch. Set AL = CX = DX = SI = 0 and DI = 1.
3. If the call in step 2 returns with the carry flag set and AX = 1, SHARE is not loaded. Move to step 5.
4. SHARE is loaded. Unlock the region of the file using MS-DOS Interrupt 21h Function 5Ch. Set CX = DX = SI = 0 and AL = DI = 1.
5. Close the file using MS-DOS Interrupt 21h Function 3Eh.
6. Delete the file using MS-DOS Interrupt 21h Function 41h.

Note that the drive on which the temporary file is created is important in a network environment. Typically, SHARE is always loaded for network drives; however, a copy of SHARE is running on the server, not on the workstation. Therefore, the application should run the test above against the drive(s) from which it will access files.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrFileioShare.exe

FIX: SetHandleCount() Causes UAE or Hang

Article ID: Q74512

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9107007

SYMPTOMS

When an application calls the SetHandleCount function to increase the number of available file handles, the application experiences an unrecoverable application error (UAE) or the machine hangs. This fault generally occurs when the wNumber parameter is set to 80 or more.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0 where there is no way to work around this problem. Do not try to use Interrupt 21h function 67h (set handle count) because it does not work correctly in Windows 3.0.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrFileioSethandle

BUG: OpenFile Function Fails on Novell Temp Drive
Article ID: Q87347

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
-

SYMPTOMS

=====

In the Microsoft Windows graphical environment, when an application uses the OpenFile() function to open a file on a Novell network temporary drive, the function returns the value HFILE_ERROR to indicate failure.

CAUSE

=====

The OpenFile() function does not properly parse the nonalphabetic characters that the Novell network redirector uses to represent temporary drive mappings.

RESOLUTION

=====

To work around this problem, use the _lopen() function to obtain a file handle.

STATUS

=====

Microsoft has confirmed this to be a problem with Windows version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrFileioFileapi

SAMPLE: Reading the Boot Sector of a Drive

Article ID: Q102870

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1
-

SUMMARY

=====

BOOTSEC demonstrates how to use Interrupt 25h (absolute disk read) to read the boot sector (the first sector on head 0, cylinder 0) off of a drive (either a floppy disk drive or hard disk).

BOOTSEC checks to see whether the drive is one of the following:

Drive	Detection Method
CD-ROM	Interrupt 2F calls to MSCDEX.
Net drive	Windows API WNetGetConnection().
RAM drive	Checks the boot sector to see if there is one FAT.
Hard disk	Checks the media BYTE of the boot sector. If it is equal to 0xF8h then it is a hard disk.
Floppy disk	Checks the media BYTE of the boot sector. If it is not equal to 0xF8h and it is not a RAM drive, net drive, or CD-ROM drive, then it is a floppy disk drive.

BOOTSEC also shows how to implement a dialog box as a main window using a private dialog class.

MORE INFORMATION

=====

The following information is contained in the boot sector:

- The jump instruction to the boot strap routine
- The name of the OEM and the version of MS-DOS
- Bytes per sector
- Sectors per cluster
- Reserved sectors
- Number of file allocation tables
- Number of root directory entries
- Number of sectors
- Media descriptor
- Number of sectors occupied by each FAT
- Number of sectors on a single track
- Number of read/write heads on the drive
- Number of hidden sectors
- Number of huge sectors
- Whether the disk is the first hard disk drive
- The boot signature
- The volume serial number

- The volume label
- The file system type

NOTE: The information contained in the boot sector was changed in MS-DOS 5.0. If this program is run on a disk that was formatted with a previous version of MS-DOS, then some of the fields in the structure will not be filled out, and the program may display garbage. The elements of the structure that were not changed will be displayed correctly.

BOOTSEC can be found in the Software/Data Library by searching on the word BOOTSEC, the Q number of this article, or S14214. BOOTSEC was archived using the PKware file-compression utility.

Additional reference words: 3.10 INT

KBCategory:

KBSubcategory: KrFileioMisc

PRB: File Handles Cannot Be Shared Between Programs or DLLs
Article ID: Q46524

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

Assume that there are two applications, A and B. Application A calls a Dynamic-Link Library (DLL) to open a file. The file pointer (FILE * pFile) is stored on the data segment of the DLL. Application A then calls a function in the DLL to read the record "n" of this file properly. However, if Application B calls the same function in the DLL to read the same record, the record appears as random characters.

RESOLUTION

File handles cannot be shared between applications or DLLs. Each application has its own file handle table. When an OpenFile call is made, a file is taken out of the application's program segment prefix (PSP). For two separate applications to use the same file, each application must make its own OpenFile call, file I/O calls, and close.

Additional reference words: 2.03 2.10 2.x 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrFileioFileshare

INF: Failure to Load Resources When All File Handles Are Used
Article ID: Q50741

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

A Windows application should not use all available file handles. Doing so may prevent Windows from being able to load the application's resources.

When Windows has to open the application's .EXE file to retrieve a resource (such as the icon's bitmap), it uses one of the application's file handles. If the application has used all available file handles, Windows cannot load the resource.

For example, suppose LoadIcon() was called previously to obtain an icon handle successfully, and the icon is being used as a window's icon. The rendering of the icon will fail if the application is using all the file handles. For example, if the window is to be minimized, the icon will be displayed as a black block on the screen.

Note: LoadIcon() loads in the logical information of the icon; the bitmap of the icon is not loaded until it is going to be used.

In Windows 2.11 (retail and debug versions), failure to load an icon's bitmap due to lack of available file handles will hang the system.

Under MS-DOS, an application has 20 file handles when it begins executing. Five of these handles (that is, STDIN, STDOUT, STDERR, and STDAUX) are automatically opened for use by the operating system. This leaves a total of 15 file handles available for an application. If a Windows application needs more file handles open at any given time, it can use the SetHandleCount() API.

Additional reference words: 2.03 2.10 3.00

KBCategory:

KBSubcategory: KrFileioSethandle

INF: Do Not Use the MS-DOS APPEND Utility in Windows

Article ID: Q58412

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The MS-DOS APPEND utility remaps the contents of specified directories into the current directory, which makes the files of these directories available to an application. Do not use this utility on systems running the Windows operating environment because the APPEND utility is fundamentally hostile to the operation of Windows. The design of Windows rests on its ability to build a fully-qualified path for each file it opens.

More Information:

When Windows opens an application file (for example, WINWORD.EXE), it stores the fully-qualified path of this file (for example, D:\WINWORD\WINWORD.EXE). With this information, Windows can reopen the file even if user or application activity changes the current drive and current directory.

The problem with the APPEND utility is that it prevents Windows from reliably determining the proper fully-qualified path to a file. If an application calls the open function when C:\EXCEL is the current drive and directory, and the D:\WINWORD directory is specified in the APPEND search path, Windows may improperly record the fully-qualified path to the WINWORD.EXE file as follows: C:\EXCEL\WINWORD.EXE.

In this situation, when Windows reopens the file later, it receives an error from MS-DOS because the file is not actually located in the drive and directory indicated by the stored fully-qualified path. When Windows detects this error, it displays the Change Disk message box.

The APPEND utility can cause similar problems for the WINOLDAP module, which runs MS-DOS (non-Windows) applications under Windows. These problems can result in unexpected "File Not Found" errors, failure to start an MS-DOS application, failure when the MS-DOS application exits, or failure when the user tries to switch back to Windows.

The current versions of all the major application software available today do not require the APPEND utility. The MS-DOS version 4.0 and 4.01 installation programs usually add the APPEND utility to the user's configuration, which consumes valuable application memory without providing any benefits to the end user. The APPEND utility is usually found in the AUTOEXEC.BAT file. The line with APPEND can be removed or commented out by placing the word "remark" at the beginning of the line.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrFileioMisc

INF: Incomplete Description of SetErrorMode() Function
Article ID: Q100305

Summary:

The description of the SetErrorMode function does not list the flag to reset the default behavior of Windows; that is, to display all the error message boxes.

More Information:

On page 840, the Microsoft Windows Software Development Kit (SDK) "Programmer's Reference, Volume 2: Functions" manual for version 3.1 lists the following three flags for the SetErrorMode function:

```
SEM_FAILCRITICALERRORS  
SEM_NOGPFAULTERRORBOX  
SEM_NOOPENFILEERRORBOX
```

These flags can be combined to prevent the display of message boxes for critical error faults, general protection (GP) faults, and file-not-found errors, respectively. However, the SetErrorMode function description doesn't list the default flag that is used to display all the error message boxes.

To allow the SetErrorMode function to display all the error message boxes, pass a zero as the fuErrorMode parameter.

Additional reference words: 3.1 3.10 INT 24h File-I/O

KBCategory:

KBSubcategory: KrFileIomisc

BUG: OpenFile Fails When UNC Server Name Longer than 11 Chars
Article ID: Q101414

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
-

SYMPTOMS

=====

OpenFile() successfully opens files with universal naming convention (UNC) names when the server portion of the name is 11 characters or shorter, but fails to open files when the server name is longer than 11 characters. Error code number 2, "File not found," is placed in the nErrCode member of the OFSTRUCT structure passed to OpenFile.

CAUSE

=====

OpenFile() validates filenames before opening them, and in the case of UNC names, allows server names that are 11 characters or shorter only. No attempt is made to open a file with a UNC name longer than 11 characters.

RESOLUTION

=====

To open files with UNC names when the server name is longer than 11 characters, use _lopen().

STATUS

=====

Microsoft has confirmed this to be a problem in the Windows SDK version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.10 buglist3.10 WFW NETWORK

KBCategory:

KBSubcategory: KrFileioFileapi

INF: Sharing Files with Windows for Workgroups Clients
Article ID: Q101421

The information in this article applies to:

- Microsoft Windows Software Development Kit version 3.1
 - Microsoft Windows for Workgroups version 3.1
-

SUMMARY

=====

Multiple applications, each running in Windows for Workgroups version 3.1 on different workstations, can share a file residing on a server with full read-write access while simultaneously keeping the file open. File region locking ensures that data integrity of the file is not lost. The Windows for Workgroups network redirector caches file I/O operations unless file locking is used. It will seem that Windows for Workgroups clients are failing to write data unless the applications lock file regions.

MORE INFORMATION

=====

Opening a file with READ_WRITE access and using the OF_SHARE_DENY_NONE file sharing flag is possible from more than one application each running from a different workstation. Applications sharing a file from a server must use the same file sharing flags. Keeping the file open may be a necessary specification for networked database management systems because it is more efficient to keep the file open than to open and close the file for each file-access operation. However, not all applications should be designed to keep the file open during the use of a file.

To better understand file sharing, consider an application running on Workstation A opening a file that resides on the disk of Server S. An application running on Workstation B can also open the same file so that each application is aware of the updates made to the file on Server S if the file is opened with READ_WRITE access and the OF_SHARE_DENY_NONE sharing flag.

To ensure that two or more applications do not access the shared file in the same region at the same time, locking is used to create exclusive access to a region of the file. File region locking by one application causes failure of file I/O operations performed by other applications sharing the file. The sharing component of Windows for Workgroups, VSHARE.386, keeps track of the region that is locked and fails the file I/O operation attempting to access the shared file, including any portion of the locked region. This is very critical to file sharing; if one application is writing information into a location of a shared file at the same instance another application is reading from the same location of the shared file, there is a loss of data integrity.

In Windows for Workgroups, file I/O operations are redirected to the server drive via the network redirector component, VREDIR.386 (in enhanced mode) or WORKGRP.SYS (in standard mode). The network redirector of Windows for Workgroups caches file I/O operations. The redirector will disable the cache if file locking is enabled, and therefore even if two applications are sharing files and are being careful so that there is no file access concurrency, Windows for Workgroups will still fail to perform the file I/O unless file locking is used. One application reading from the file will not be reading the latest updates from the application writing to the file or data will not seem to be written to disk.

The following summarizes key points when implementing applications to share files residing on a server for full read-write access.

- Open the file with READ_WRITE access and OF_SHARE_DENY_NONE sharing flag.
- Before reading or writing, lock the region of the file about to be accessed. Use _locking() from the Microsoft C run-time library or MS-DOS Interrupt 21h function 5Ch to lock the file region.
- Immediately following a read or a write, unlock the region. Use _locking() or Interrupt 21h function 5Ch to unlock the file region. For courtesy to other applications using the file, do not exclude the use of a file region for a long period of time. Lock the file region only during accessing (reading or writing).
- If the lock fails, this means that some portion of the region of the file is locked by another application. The application is not allowed access to this region until the other application releases it by unlocking the region.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrFileioFileshare

PRB: File Attributes/Date/Time Fail to Set on Open File
Article ID: Q102554

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) versions 3.0 and 3.1
 - Microsoft Windows for Workgroups version 3.1
-

SYMPTOMS

=====

When using MS-DOS function 5701h (set file date and time) along with function 4301h (set file attribute) while the file is currently open, the date and time fail to be set without error. This problem occurs when the file resides on a drive that is shared by Windows for Workgroups version 3.1.

RESOLUTION

=====

To work around the problem, the application should set the time and date of the file while the file is open, close the file, and then set the file attribute.

STATUS

=====

Microsoft has confirmed this to be a problem in Windows for Workgroups version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

=====

This operation is commonly performed by copy-file routines implemented to preserve the time and date stamp as well as the file attributes of the source file when creating the destination file. The copy-file routine can read the file time, date, and attributes of the source file and then set the same on the destination file. If the file is being created on a network shared drive that is served by Windows for Workgroups version 3.1, the problem mentioned above will occur. The problem does not occur if only the file time and date are being set or if only the file attributes are being set while the file is still open. There is no reason to keep a file open when setting the attributes because the function refers to the file by a string containing the filename, and not by a file handle.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrFileioAttribs

INF: The
Article ID: Q102640

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

SUMMARY

=====

Running an application in the Microsoft Windows operating environment may produce a system modal dialog box containing the following error message:

Segment Load Failure

The error occurs inconsistently due to the cached file handle mechanism of Windows version 3.1.

MORE INFORMATION

=====

During the life of a Windows application, Windows may need to load discarded code or resource segments or LOADONCALL segments for the first time. To load the segment, Windows uses file I/O functions to read the information from the .EXE file. If Windows has any errors opening the .EXE file, it will produce the error message:

Segment Load Failure

Possible reasons opening the .EXE file fails are:

- The filename was changed, deleted, or otherwise corrupted since the application was first started.
- The .EXE file resides on a shared drive and is opened exclusively or in compatibility mode by another workstation running the application.
- The .EXE file resides on a shared drive and is opened exclusively or in compatibility mode by another application for some purpose other than running the .EXE.
- The system has run out of available file handles.

The error message may occur inconsistently or not at all due to the mechanism used by Windows for caching file handles. The .EXE file is referred to by a file handle. By default, Windows caches 12 file handles for the most recently used files. If Windows subsequently opens more files than what the cache can hold, the cache closes the least recently used file handles. The error does not occur if Windows uses a cached file handle to refer to the file, but will occur if the file handle is no longer in the cache.

The file handle cache is a system-wide mechanism and not a

per-application mechanism. The number of cached file handles can be changed by the `CachedFileHandles` switch in the `[boot]` section of the `SYSTEM.INI` file. For more information regarding `CachedFileHandles`, see the `SYSINI.WRI` file in the Microsoft Windows Resource Kit.

Additional reference words: 3.10

KBCategory:

KBSubcategory: `KrFileioMisc`

INF: How OF_SHARE Modes Affect Opening Files

Article ID: Q104498

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
-

The following modes have an effect only if SHARE is loaded:

OF_SHARE_COMPAT (same as specifying no sharing flags, thus its name): If the file has not been opened in any other sharing mode, the current task may open the same file in OF_SHARE_COMPAT mode any number of times. If the current task attempts to open the file in any other sharing mode, the open will fail. If other tasks attempt to open the file with any sharing mode, the open will fail. If the file has the read-only file attribute, other tasks may open the file in OF_SHARE_COMPAT mode any number of times.

OF_SHARE_DENY_WRITE: If the file has not been opened in any other sharing mode, any task may open the file any number of times in OF_SHARE_DENY_WRITE. Attempting to open the file in any other sharing modes will fail.

OF_SHARE_DENY_READ: If the file has not been opened in any other sharing mode, any task may open the file any number of times in OF_SHARE_DENY_READ. Attempting to open the file in any other sharing modes will fail.

OF_SHARE_EXCLUSIVE: If the file has not been opened in any other sharing mode, the file will be opened. Any additional attempts to open the file in any sharing modes by any task will fail.

OF_SHARE_DENY_NONE: If the file has not been opened in any other sharing mode, any task may open the file any number of times in OF_SHARE_DENY_NONE. Attempting to open the file in any other sharing modes will fail.

Additional reference words: 3.00 3.10 share file open dos

KBCategory:

KBSubcategory: KrFileioFileshare

INF: Windows Code Module to Delete Files

Article ID: Q96789

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The DELTEST sample contains a complete code module to delete one or more files by using wildcards. The sample also demonstrates how to use wildcards to search through an entire directory.

DELTEST can be found in the Software/Data Library by searching on the word DELTEST, the Q number of this article, or S14123. DELTEST was archived using the PKware file-compression utility.

More Information:

The C run time contains a function to delete one file, but does not allow a program to delete multiple files using wildcard characters. The DELETE.C module included with this sample can be used with any application or dynamic-link library (DLL) to perform this operation. This module uses the `_dos_findfirst/next` functions in conjunction with the `remove()` function to allow multiple files to be erased.

Note: In the sample, if "Delete tmp Files" is selected from the menu, the two .TMP files included with the sample will be erased.

For information on how to call the `Delete()` function, please refer to the comment block inside of DELETE.C.

Additional reference words: 3.x DelTest DELETE.C

KBCategory:

KBSubcategory: KrFileioFileapi

PRB: DLL Function Returns Float or Double Value Incorrectly
Article ID: Q86081

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

When a function using the C calling convention and exported from a dynamic-link library (DLL) returns a data value of type float, double, or long double, the calling application receives unexpected values. This behavior occurs in applications developed with the Microsoft Windows Software Development Kit (SDK) version 3.0 or 3.1 or with version 1.0 of Microsoft QuickC for Windows.

CAUSE

The pointer used to return a floating-point result under the C calling convention is invalid once control returns to the application.

RESOLUTION

Declare the DLL function using the Pascal calling convention or allocate memory from the global heap to hold the floating-point result and return the handle from the DLL function.

More Information:

The DLL and any application that calls the DLL each have separate floating-point accumulators. When an application calls a DLL function declared with the Pascal calling convention, the application allocates space on the stack to receive the returned data type. The DLL function pushes the value onto the stack for the application to use.

When an application calls a DLL function that uses the C calling convention, no stack space is allocated because the calling function cleans up the stack. Under the C calling convention, the DLL function returns a pointer (in DX:AX) to the floating-point accumulator, which contains the result. However, once the application regains control, the pointer is not valid.

The code examples below demonstrate returning a float value under the C and Pascal calling conventions:

C Calling Convention

```
// C calling convention - DLL  
// Compile options required: /Asw /G2sw /Zp  
  
HANDLE _far floatcalc(float f11, float f12);
```

```

HANDLE _far floatcalc(float f11, float f12)
{
    HANDLE hFloat;
    float _far *pFloat;

    hFloat = GlobalAlloc(GMEM_MOVEABLE, sizeof(float));
    pFloat = (float _far *)GlobalLock(hFloat);
    *pFloat = f11 * f12;
    GlobalUnlock(hFloat);
    return hFloat;
}

// C calling convention - Application
// Compile options required: /AS /G2sw /Zp

extern HANDLE _far floatcalc(float f11, float f12);

void Calc(void)
{
    float _far *pFloat, f1;
    HANDLE lFloat;

    lFloat = floatcalc((float)3.0, (float)4.1); // Call DLL function
    pFloat = (float _far *)GlobalLock(lFloat);
    f1 = *pFloat;
    GlobalFree(lFloat);
}

```

Pascal Calling Convention

```

// Pascal calling convention - DLL
// Compile options required: /Asw /G2sw /Zp

float _far _pascal FloatCalc(float f11, float f12);

float _far _pascal FloatCalc(float f11, float f12)
{
    return f11 * f12;
}

// Pascal calling convention - Application
// Compile options required: /G2sw /Zp

extern float _far _pascal FloatCalc(float f11, float f12);

void Calc(void)
{
    float temp;

    temp = FloatCalc((float)3.1, (float)4.2); // Call DLL function
}

```

Additional reference words: qcw qcwin 1.00 3.00 3.10

KBCategory:

KBSubcategory: KrFltptMisc

INF: Applications and the Math Coprocessor Under Windows
Article ID: Q43276

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

When an application for Microsoft Windows is run on a machine with a math coprocessor, the application can use inline floating-point instructions to take the fullest advantage of the hardware.

More Information:

Specifying the `-FPi` option on the C compiler command line causes the Microsoft C optimizing compiler to produce inline 80x87 math coprocessor code for any floating-point math operation. If this code is linked with the `WIN87EM.LIB` library and a math coprocessor is present in the system at run time, the application will use the inline floating-point instructions for its math operations. If no coprocessor is available at run time, code in the emulator library evaluates floating-point expressions.

An application compiled for the MS-DOS environment with the `-FPi` option checks for the coprocessor at run time and modifies its code accordingly: if there is a coprocessor, it uses inline floating-point instructions; if there is no coprocessor, it calls software routines to emulate the coprocessor.

In the Windows environment, these run-time modifications are not performed because the Windows kernel fixes up the floating point references as it loads the application's code segments (the kernel is aware of the presence or absence of the numeric coprocessor). This means that an application for the Windows environment compiled with the `-FPi` option will perform direct, inline floating-point instructions without run-time coprocessor-checking. Consequently, there is no need to link in the Microsoft C Compiler inline floating-point module, which removes the run-time coprocessor-checking.

Additional reference words: 1.03 1.04 2.03 2.10 3.00 3.10 1.x 2.x

KBCategory:

KBSubcategory: KrFltptCmplropts

INF: EMS Support in Windows Version 3.00 and 3.10

Article ID: Q57954

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The information listed below discusses the following topics:

1. Windows version 3.x support of EMS.
2. Utilization of the services of an already-installed LIM EMS (Lotus/Intel/Microsoft Expanded Memory Support) version 4.0 driver, such as a "limulator" or hardware EMS board by Windows version 3.x.
3. Making EMS calls from a non-Windows application.
4. Making EMS calls from a Windows application.

More Information:

1. Windows version 3.x support of EMS.

Windows version 3.x has three modes: real mode, standard mode, and enhanced mode. Of the three, only enhanced mode provides EMS. This is because "limulation" [simulation of LIM (Lotus/Intel/Microsoft)] requires hardware support that only the Intel 80386 provides; the 8086 and 80286 do not have the necessary combination of protected mode addressing and virtual 8086 emulation (provided by the 80386's V86 mode).

2. Utilization of the services of an already-installed LIM EMS (Lotus/Intel/Microsoft Expanded Memory Support) version 4.0 driver, such as a "limulator" or hardware EMS board by Windows version 3.x.

Real mode Windows can take advantage of either a limulator or a hardware EMS board.

Windows version 3.0 standard mode (which runs in 80286 protected mode) cannot use a limulator because these drivers run in 80386 protected mode, thereby conflicting with the standard mode's need to switch into protected mode. However, if the machine has a hardware EMS board and associated LIM version 4.0 driver installed, any EMS calls made will be serviced by the EMS driver; however, Windows itself will not use the EMS memory in any way. Only small frame EMS is supported in protected mode.

In Windows version 3.1, it is possible to start limulators such as QEMM, 386MAX and EMM386 in standard mode. All support for EMS from protected mode Windows applications has been dropped from Windows

version 3.1 enhanced mode. Any INT67H calls made by a Windows application in enhanced mode under Windows version 3.1 will result in an error being returned.

Enhanced mode is incompatible with preinstalled EMS drivers that cannot be "shut off" by Windows. If the EMS driver is a limulator that cannot be shut off, Windows will not be able to run in enhanced mode because the 80386 is already running in protected mode; limulators that can be shut off respond to Windows as it boots, turning themselves off and letting Windows manage the EMS that they had been managing.

Physical LIM memory is not supported by 386 enhanced mode Windows. Software running in Virtual 8086 mode in any VM (virtual machine), including the system VM (which contains all the Windows applications), can make LIM version 4.0 calls; however, the calls will be handled by Windows/386, not by any previously installed LIM driver. No LIM calls are reflected; they are all handled (error or successful completion) by V86MMGR. As mentioned above, only small frame EMS is supported in protected mode.

3. Making EMS calls from a non-Windows application.

If EMS is present, non-Windows applications have the full LIM version 4.0 specification available for their use. Running in enhanced mode, part of the mappable page array may not be usable; therefore, applications have to be sensitive to what is usable in the mappable page array and what is not; they cannot assume that all of the mappable page array is available.

4. Making EMS calls from a Windows application.

If EMS is present, the normal small-frame EMS calls work as they do in Windows version 2.x (that is, applications can make LIM version 3.2 calls plus LIM version 4.0's realloc function). This allows both Windows versions 2.x and 3.x applications to use EMS memory.

CAUTION: In protected mode, the standard technique of querying for the presence of an EMS driver by examining the contents of interrupt vector 67H will fail due to virtualization of memory and virtualization of the IDT (interrupt description table). Therefore, applications will have to use the alternate technique of opening a file handle using the name of the EMS device driver. For more information on these techniques, please consult Chapter 9 of Ray Duncan's "Advanced MS-DOS" book (Microsoft Press, 1986), which is the chapter on memory management under MS-DOS. This chapter contains a discussion on EMS that describes how to check for the presence of an EMS driver.

Windows version 3.x applications that use EMS when running in protected mode will be slower than non-EMS-using applications, and may run out of memory faster. If it is necessary to run a Windows version 2.x application that requires EMS under Windows version 3.x, this can easily be done in enhanced mode. When designing a new application for Windows version 3.x, however, the programmer should take advantage of the new protected mode addressing scheme rather than trying to use EMS.

CAUTION: In protected mode, the value returned as the location of the page frame is a selector, not a segment. Therefore, applications cannot make any assumptions about that value. Many EMS-using applications, besides encountering the inefficiency of using EMS, will probably run into other restrictions that will not allow them to run in protected mode. For example, they cannot put code into EMS unless they implement special handling in protected mode to create a code selector alias for the page frame selector.

For enhanced mode, there are SYSTEM.INI entries that correspond to the same entries in the .PIF file. The Control Panel does not touch them. We do not expect many programmers to need to do anything with these at all. See the PIFEDIT documentation for information on the meanings of possible values.

Example:

```
; The following variables perform the equivalent of the XMS and
; EMS memory PIF settings for the SYS VM:
;
SYSVMEMSLIMIT           Max value (default is -1)
SYSVMEMSREQUIRED       Min value (default is -1)
SYSVMEMSLOCKED         Is the EMS touched under interrupt or needed
                       to be in memory at all times for some other
                       reason? (default is NO)
SYSVMXMSLIMIT          Max value (default is -1)
SYSVMXMSREQUIRED       Min value (default is -1)
SYSVMXMSLOCKED         Is the EMS touched under interrupt or needed
                       to be in memory at all times for some other
                       reason? (default is NO)
```

Related switch:

```
EMMSIZE                Controls the maximum amount of memory
                       that will be used for EMS across all
                       VMs (default is -1, no limit).
```

Additional reference words: 2.03 2.10 2.x 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrIsrtsrDosdrvs

INF: How to Get a Pointer to the Stack

Article ID: Q11941

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The following information applies to Windows versions 2.x and 3.0.

C pushes its parameters in reverse order (right to left) for the `_cdecl` calling convention (default) and in order (left to right) for the Pascal calling convention.

The first (or last, for the Pascal convention) parameter is always the last one pushed, and always has the same address relative to the start of the frame.

In the C compiler documentation, see `vsprintf` and the `va_start` macro in `STDARG.H` for an example of accessing variable arguments on the stack.

To get a pointer to the stack, use the following code:

```
far MyFunction()
{
    int x;
    int far *y = &x;
}
```

Note that this pointer will not be valid when the function is exited, since the stack contents will change.

Additional reference words: 2.0 2.1 3.0

KBCategory:

KBSubcategory: KrMmMisc

INF: How Windows Resolves Far Calls When Movable Flag Is Used
Article ID: Q11979

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The following information applies to Windows versions 2.0, 2.03, and 2.1. This information only applies to Windows versions 3.0 and 3.1 in real mode.

The information listed below addresses the following topics relating to using the middle model of compilation (-AM) when a module's code segment has been renamed using the -NT switch, and the segment is declared movable in the module definition (.DEF) file:

1. Locking of the code segment by Windows
2. The handling of FAR calls

Windows does not necessarily keep the code segment locked. The Microsoft C Compiler uses the BP register as a "frame pointer". Local variables and parameters are always accessed using offsets from the BP register. The BP register is initially even, and the Windows stack is word aligned. When a FAR call is made, BP is increased by one. If the code segment is discarded, the stack is walked and patched. By determining if BP is odd or even, Windows can tell whether the call is FAR or NEAR.

When a long return address is on the stack, it has a pushed DS and BP, and because the BP register is increased by one for FAR frames, FAR frames may be detected by walking the task chain and BP stack-frame chains.

When the Windows prolog is set up, it does the following:

```
extern far pascal funcname();

cProc funcname,<FAR,PASCAL>
"cBegin"
Prolog: push ds ; Fixed/Moveable Multiple Data Segment Support
pop ax
nop
inc bp ; Far Frame Marker/Moveable Code Support
push bp
mov bp,sp
push ds ; Data Context Switch Code
mov ds,ax ; "

...
```

```
"cEnd"  
Epilog: sub bp,2  
mov sp,bp  
pop ds ; Data Context Switch Code  
pop bp  
dec bp ; Far Frame Cleanup  
ret
```

Additional reference words: TAR56803 2.00 2.03 2.10 2.x 3.00 3.10

KBCategory:

KBSubcategory: KrMmRealmode

INF: Global Lock Count Changes in Windows 3.x

Article ID: Q61285

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The global lock count mechanism has been changed in the Windows version 3.x protected modes (that is, standard mode and enhanced mode). The GlobalLock function only affects the lock count of discardable objects and the default data segment (DGROUP); movable objects are not affected. Thus, repeated calling of the GlobalLock and GlobalFlags functions on a GMEM_MOVEABLE object does not show any changes to its lock count.

More Information:

The following are reasons and explanations concerning this design change:

1. In real mode, the GlobalLock function fixes the segment:offset of a global memory object. It also increases the lock count, as reported by the GlobalFlags function.
2. In protected mode, the far pointer returned by the GlobalLock function is a selector:offset, not a segment:offset. Because the selector value does not change, the GlobalLock function does not actually fix the memory object in the physical address space. Thus, the GlobalLock function in protected mode does not change the lock count, unless the object is discardable or is a default data segment.

In the case of a discardable object, the lock count is meaningful, because Windows needs to know when the object can be discarded (which is when its lock count is zero).

3. However, some applications have used the GlobalLock lock count as a "reference count" [that is, as an indication of how many times the GlobalLock function was called]. If the lock count for an object goes to zero, these applications might consider the object a candidate for being manually discarded, perhaps after copying the data to disk.

Unfortunately, this use of the GlobalLock function as a reference count keeper does not work in protected mode. Applications that symmetrically pair calls to the GlobalLock function with calls to the GlobalUnlock function do not need to know the lock count, and therefore, are unaffected by this change in behavior.

4. How does an application keep track of reference counts now, given

that the GlobalLock approach does not work for nondiscardable objects in protected mode? The application should really keep track of reference counts itself, which should not be hard to do because the application in need of this functionality will have a table of global handles anyway.

However, if the application cannot be modified to maintain its own reference counts, then there is a new Windows function, called GlobalFix, that will accomplish this functionality. The GlobalFix function performs the following functionality:

- a. It fixes the object in the protected mode linear space.
 - b. It increments the "lock count", as returned by the GlobalFlags function.
5. The following includes more information about the GlobalFlags function. In real mode, it returns the GlobalLock lock count. In protected mode, if the object is discardable, the GlobalFlags function also returns the GlobalLock lock count. In protected mode, if the object is nondiscardable, the GlobalFlags function returns the GlobalFix reference count. In other words, the GlobalFlags function always returns the lock/fix count. However, in protected mode, the GlobalLock and GlobalUnlock functions do not affect the count, only the GlobalFix and GlobalUnFix functions do.

Note: In real mode, the GlobalFlags lock count actually indicates the sum of GlobalLock's and GlobalFix'es. Therefore, if the programmer is calling GlobalLock's and GlobalFix'es in pairs, then the GlobalFlags lock count actually is twice the logical reference count, if in real mode.

6. If the application needs to keep track of reference counts, and the programmer wants Windows to do the work for them, then the programmer must accompany every call to the GlobalLock function with a call to the GlobalFix function. This way the programmer will be able to depend on the validity of the GlobalFlags' lock/reference count. However, using the GlobalFix function just to keep track of the reference count is overkill, if that is all the programmer wants it to do. Remember, the GlobalFix function also fixes the object in the protected mode linear address space. The price the programmer pays for having Windows keep track of the reference count (by using the GlobalFix function) is the following:
- a. Every time the GlobalLock or GlobalUnlock function is called, the programmer must also call either the GlobalFix or GlobalUnFix function.

And, much worse:

- b. The programmer establishes sandbars in the linear address space.

An application should either keep track of reference counts on its own or always pair GlobalLock calls with matching GlobalUnlock calls; the use of the GlobalFix function should be avoided.

Very few applications should need to fix global objects in linear space; therefore, few applications should need to use the GlobalFix function.

7. If the programmer needs to unconditionally unlock and free a global memory object of any type, then code similar to the following can be used:

```
/* 1. Make it discardable if necessary. */
if (GlobalFlags shows that it is nondiscardable)
    GlobalRealloc (GMEM_MODIFY it to be discardable);

/* 2. Remove any lock counts that might be on it. */
while (GlobalUnlock != 0) /* keep unlocking it */
    ;

/* 3. Free it. */
GlobalFree()
```

Additional reference words: 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrMmFixlockwire

INF: WINMEM32 Not Version Dependent

Article ID: Q73666

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

WINMEM32 is a dynamic-link library (DLL) designed to run in Microsoft Windows enhanced mode to provide support for 32-bit flat memory model code under Windows. While WINMEM32 is not bound to a particular version of Windows, it does require enhanced mode.

New versions of WINMEM32 that might become available in the future should not affect the ability to use old versions of WINMEM32 on a given system. Note, however, that only one version of WINMEM32 can be loaded at a time.

Newer versions of WINMEM32 will be backward compatible with older versions. Because you should upgrade to the new version, if possible, it is very important that WINMEM32 applications do not tightly version bind to WINMEM32. An application should always use the greater-than-or-equal-to operator (\geq), never the equal-to operator ($=$), to compare the result of the WINMEM32 GetWinMem32Version function to the required version.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrMmWinmem32

INF: Heap Placement in Memory

Article ID: Q12245

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The Windows system uses `LocalInit(ds,0,size_in_bytes)` to align the heap. The first 16 bytes in DS are the NULL segment. It contains a block of "reserved pointers." These are the heap pointers. The heap is located in DS by subtracting the size of the heap from the end of DGROUP and filling in the start of the heap pointer with the resulting offset from the end of DGROUP. This procedure avoids data-stack/heap collision.

Additional reference words: 2.00 3.00

KBCategory:

KBSubcategory: KrMmLocalmem

INF: Overview of How to Share Memory Between Applications
Article ID: Q64126

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Listed below are three ways to share memory between Windows applications; these are the only methods that are guaranteed to work in all memory configurations and in future versions of Windows:

1. Using the Clipboard
2. Storing information in the data segment of a shared Dynamic-Link Library (DLL)
3. Dynamic Data Exchange (DDE)

More Information:

1. The Clipboard is the easiest method to use, and is discussed in the Windows Software Development Kit (SDK) as well as in such books as Charles Petzold's "Programming Windows."
2. Sharing data in the data segment of a DLL is possible because there is only one data segment for all instances of the DLL; DLLs are not "instanced." Because of this, it is possible to have the DLL do a LocalAlloc() out of its local heap, which is part of its DGROUP and thus is limited to 64K. Programmers must determine the memory scheme that best suits their needs and what calls they will make to the DLL to copy/share that memory to other applications that call into it.
3. DDE is designed to allow applications that follow the protocol to share/pass data back and forth. An "envelope and letter" analogy, which is listed below, provides an example of how this works:

If some information needs to be sent from one person/application to another person/application, do the following:

- a. Address the envelope: Call GlobalAlloc() on a piece of global memory with the GMEM_DDESHARE flag.
- b. Write the letter on a piece of paper: Call GlobalLock() and write to the global memory.
- c. Seal the letter: Call the GlobalUnlock() function.
- d. Send the letter off to the other person: Use the PostMessage() function with a WM_DDE_DATA message that has the hGlobalMemory in it.

To receive and read the letter, the other person/application

does the following:

- a. Get the letter: A WM_DDE_DATA message is found in the message queue, along with the handle of the global memory, hGlobalMemory.
- b. Open the envelope: Call GlobalLock (hGlobalMemory).
- c. Make a copy of the letter and read it:
 - 1) Create a new envelope: Call GlobalAlloc(hNewEnvelope) and use the GMEM_DDESHARE flag IF the letter needs to be sent back.
 - 2) Open the new envelope: Call GlobalLock(hNewEnvelope).
 - 3) Copy the contents of the old letter to the new letter, modifying the contents at this time if necessary.
 - 4) Seal the envelope: Call GlobalUnlock().

If the person/application wants to send the letter back to the person/application that originally sent the letter, perhaps with some answers to questions asked in the original letter, the following procedure should be used:

- 5) Send the letter: Call PostMessage() with the new handle to global memory.
- d. When done with the old letter, throw it away: Use GlobalUnlock() on the handle and then call GlobalFree().

According to the DDE specification, the rules for freeing the global memory object are as follows:

Receiver deletes memory unless either of the following is true:

- 1) fRelease flag is zero.
- 2) The fRelease flag is 1; however, the receiving (client) application responds with a negative WM_DDE_ACK message.

For more information, please refer to Chapter 15 in the Windows SDK reference manual.

For more information on sharing memory, please refer to Chapters 15 and 16 of the "Microsoft Windows Software Development Kit Guide to Programming."

For more information on DDE, please refer to Chapter 22 of the "Microsoft Windows Software Development Kit Guide to Programming."

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmSharedmem

INF: Accessing Physical Memory Using Kernel Exported Selectors

Article ID: Q64151

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

In Windows version 3.0, the Kernel exports a number of selectors that Windows programs can use to access commonly used hardware memory. The exported selectors include the following:

```
__0000h, __0040h, __A000h, __B000h, __B800h, __C000h,  
__D000h, __E000h, __F000h
```

The four letters after the leading "__" represent the real mode address that the selector is for. Each of the selectors has a 64K limit.

In order to use one of these selectors, implement one of the following methods:

Method 1

If the code is being written in assembler, declare the selector variable as follows:

```
extrn MySelector:ABS  
(or "externA MySelector" using C-macros)
```

Then import the selector in the definition file, as follows:

```
IMPORTS MySelector = KERNEL.__B000h
```

In the same program, if the C routines need to use the selector value, another global selector may be declared across the assembly code and the C code. Then, assign "MySelector" to the variable in the assembly code.

Method 2

Declare an extern variable in the C routine and then use the address of the extern variable as the selector. For example:

```
extern WORD _MyB000h ;  
#define MySelector (&_MyB000h)
```

Now the variable MySelector can be used to access segment B000h.

Remember to import it using the correct name (one more underscore). For example:

IMPORTS

 __MyB000h = KERNEL.__B000h

Method 3

GetProcAddress can also be called with the Kernel's module handle and the name of the selector. The lower word of the return value is the selector value.

More Information:

There are no protections on these selectors. Any other Windows program can access them at the same time. It is strongly recommended to use these selectors only in a Windows dynamic-link library (DLL). By accessing the hardware memory only through a DLL, the Windows application is given the maximum hardware independence. This method also minimizes possibilities of conflicts caused by different Windows applications trying to access the same memory at the same time.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMm

INF: Minimizing Lock and Unlock Calls in Protected Mode

Article ID: Q74197

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

When Microsoft Windows is running in protected mode (standard mode or enhanced mode), the selector of a movable memory block does not change even though Windows may move the block in physical (standard mode, enhanced mode) or linear (enhanced mode) memory.

Applications that run only in protected mode can take advantage of this behavior to minimize the number of GlobalLock and GlobalUnlock calls.

More Information:

The following code demonstrates how to allocate a global memory object:

```
HANDLE hMem;
LPSTR lpstr;

if ((hMem = GlobalAlloc(GMEM_MOVEABLE, cb)) != NULL)
    if ((lpstr = GlobalLock(hMem)) != (LPSTR)NULL)
    {
        // use the memory
    }
```

The following code demonstrates how to free the global memory object allocated above:

```
if (GlobalUnlock(hMem) != (HANDLE)NULL)
    GlobalFree(hMem);
```

Leaving movable global memory objects locked does not impose a memory consumption penalty in protected mode.

This technique is not appropriate for an application that runs in real mode under versions of Windows prior to version 3.1. In real mode, Windows manages a limited amount of memory (less than 640K) by moving and discarding memory blocks. When an application locks a memory block, Windows cannot move or discard the block. Therefore, an application that runs in real mode must lock an object only while the object is in use.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrMmFixlockwire

INF: Real Mode Not Supported by Windows 3.1

Article ID: Q78326

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

Support for real mode has been removed from Windows 3.1. Many applications designed for Windows 3.0 do not support real mode. The reasons behind this trend toward protected mode include superior memory management and smaller, faster application code.

More Information:

The remainder of this article lists the advantages and disadvantages of removing support for real mode from applications.

Advantages of Removing Real Mode Support

1. Protected mode code is smaller, cleaner, and more maintainable. These factors lead to a faster, more responsive, more reliable, system. Code is smaller and cleaner for the following reasons:
 - a. The 286 and higher processors can track memory locations in hardware, which makes locking and unlocking memory objects unnecessary. An object can be locked once when it is allocated and unlocked just prior to being freed. Because the object can move in memory even when it is locked, it is not necessary to bracket each access to an object with lock and unlock calls.
 - b. Far functions can use simplified function prolog and epilog code. For more information on this aspect of protected mode, query on the following words:

prod(winsdk) and protected and streamlined
 - c. Because protected mode code is restricted to running on 286 and higher processors, the Microsoft C Compiler -G2 switch can be used to generate smaller and faster application code.
2. Protected mode (both standard and enhanced mode) breaks the "640K barrier." Furthermore, under enhanced mode, Windows uses paged virtual memory to expand available memory by using the system hard disk as a swapping device. The large address space allows applications to have more code and data and allows users to run more applications.
3. Testing is easier because there are fewer Windows modes to test. To fully test a product that runs under real mode, five separate modes must be tested. Real mode itself contributes three of those modes: real mode with no expanded memory, real mode using the small-frame

Expanded Memory Specification (EMS) and real mode using the large-frame EMS. The other two modes are standard mode and enhanced mode. Because support for real mode has been eliminated, the same amount of testing effort can concentrate on producing a better product. It also provides an opportunity to develop and test additional enhancements. For more information on EMS, query on the following words:

prod(winsdk) and ems and developers

4. Based on a survey of Windows developers, most developers are targeting only protected mode because Windows performance on 8086-based and 8088-based machines is not satisfactory. Furthermore, these machines cannot address more than 640K of RAM.
5. "Wild writes," write-accesses to memory that incorrectly modify a memory location, can frequently be detected in protected mode through the mechanism of a GP-fault (an unrecoverable application error). It is not possible to detect these errors under real mode. These GP-faults provide information about application bugs before the application is released.

For the reasons mentioned above, Microsoft is removing support for real mode from Windows 3.1. For these same reasons, many developers have also removed support for real mode from applications developed for Windows 3.0. Applications that are written to support only protected mode should be marked with the Resource Compiler's -T switch to prevent the application from loading in real mode.

Removing support for real mode also benefits the end user because applications run faster and are more reliable. While small applications run quickly in real mode, larger applications run slowly. However, in protected mode, large applications also run quickly. A collection of large and small applications can be run simultaneously without any loss in speed. For example, when more applications are running simultaneously than can fit in the physical memory installed in the system, the paging mechanism (only available in enhanced mode) intelligently manages virtual memory to keep the most-frequently used memory pages in physical memory. This management speeds up the system.

Disadvantages of Removing Real Mode Support

1. The installed base of 8088-based and 8086-based machines cannot use the software.
2. Code that performs segment arithmetic cannot be used in protected mode. Therefore, some drivers and DOS programs that run in real mode must be rewritten for protected mode, or they cannot be run under Windows 3.1.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrMmRealmode

INF: Shrinking Heap Space

Article ID: Q21581

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

When an application calls LocalAlloc() and there is not enough memory within the application's data segment, Windows will use memory from the global heap to append this to the application's data segment. Releasing the memory that was temporarily requested then becomes an issue to the programmer.

For example, an initial HEAPWALK shows 12000 bytes free. After a 4K LocalAlloc() and LocalLock(), HEAPWALK shows 4000 bytes locked and 8000 free. Then the program allocates and locks another 10K piece; HEAPWALK shows 4000 bytes locked, 8000 free, and 10000 locked. If the program then deallocates the 4K and 10K blocks with unlock and free, HEAPWALK shows a free 12000 bytes and a free 10000 bytes. The programmer then has the problem of releasing the second free 10K block.

Using the LocalShrink() function will compact and shrink the data segment to the smallest size possible. LocalShrink() cannot move FIXED or locked blocks when compacting the local heap. Therefore, there may still be free space in the heap, and the size of the heap may not be as small as requested after calling LocalShrink(). However, this function will compact as much as possible, given this constraint.

Additional reference words: 2.00 3.00

KBCategory:

KBSubcategory: KrMmLocalmem

INF: Speed Differences Between WIN /3, WIN /2, and WIN /R
Article ID: Q49732

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The following information describes why WIN /3 and WIN /R run slower than WIN /2.

WIN /3 is slower than WIN /2 because it runs in 80386 protected mode and virtualizes all devices and I/O operations. WIN /3 also provides demand-paged virtual memory using the hard disk to swap paged-out RAM; this feature involves extra validity checking and disk swapping when page faults occur, which takes some extra time.

WIN /R is slower because it has limited memory space (the 640K maximum provided by MS-DOS), and therefore, must perform a lot of moving and discarding of code and data segments when the memory space is "overcommitted" by the system and running applications.

WIN /2 is fastest because although it runs in the protected mode of the 80286 and is somewhat slowed down by that, it has MUCH more memory available for applications to run in; therefore, less moving and discarding of code and data segments is necessary. The speed gains of WIN /2 compared to WIN /R depend on how much extended memory is on the machine and how much is needed by the system and all running applications. For example, where WIN /R has to discard and move some segments to make room for a new application to run, WIN /2 simply allocates more of the global heap, which consists of all available extended memory.

Some of these factors apply to Windows versions 2.x (2.03, 2.1, 2.11) as well as Windows version 3.0. For example, Windows/386 versions 2.x run in protected mode and virtualize devices and I/O, which slows things down. However, Windows/286 versions 2.x run in real mode and gain speed from that. On the other hand, Windows/386 versions 2.x provide Lotus/Intel/Microsoft Expanded Memory Specification (LIM EMS) version 4.0 to applications, which allows more applications to fit in memory at the same time. However, Windows/286 does not provide its own EMS memory for applications, instead relying on an already-installed EMS board or "limulator" to provide it. If the system does not have any EMS memory, Windows/286 is forced to discard and move memory more often.

KBCategory:

KBSubcategory: KrMmMisc

FIX: GlobalReAlloc() Fails in Enhanced Mode

Article ID: Q66366

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9012024

SYMPTOMS

The GlobalReAlloc() function fails to allocate requested memory even when there is enough memory available in the system to satisfy the request.

RESOLUTION

Create a function, MyGlobalReAlloc(), to use for this purpose. MyGlobalReAlloc() should perform four steps:

1. Call the system GlobalReAlloc() function to perform the operation. If the system function succeeds, the operation is complete.
2. If the system function fails, call GlobalAlloc() to allocate a new memory block.
3. If GlobalAlloc() succeeds, copy the contents of the old memory block to the new block and free the old block.
4. If GlobalAlloc() fails, there is no memory available in the system to satisfy the request.

Microsoft has confirmed this to be a problem in Windows version 3.00. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 3.0

KBCategory:

KBSubcategory: KrMmGlobalmem

INF: Validating Local Handles

Article ID: Q80864

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In Windows version 3.0, the local memory manager functions `LocalFlags()`, `LocalLock()`, `LocalReAlloc()`, `LocalSize()`, and `LocalUnlock()`, are documented to return NULL if passed an invalid handle. However, under Windows 3.0, passing an invalid handle to one of these functions can result in a non-NULL return value or an unrecoverable application error (UAE).

This article demonstrates a technique that an application can use to avoid this type of error.

More Information:

Although Windows does not automatically verify local memory handles, it is possible to verify structures in the local heap by placing a verification number in the structure. To verify the structure, check the value of this special number. The following code demonstrates this technique:

```
// Declare the structure

#define VERIFY 47
typedef struct tagDataStruct {
    int nData;           // Declare the data
    int nVerify;        // Declare the verification number
} DataStruct;

// Allocate a structure

HANDLE AllocStruct(WORD wFlags)
{
    HANDLE hLMem;
    DataStruct *pLMem;

    // Allocate the structure
    if (!(hLMem = LocalAlloc(wFlags, sizeof(DataStruct))))
        return 0;

    if (pLMem = (DataStruct *) LocalLock(hLMem))
    {
        pLMem->nVerify = VERIFY;
        LocalUnlock(hLMem);
        return hLMem;
    }
}
```



```

    return 0;
}

// Verify a handle to the structure

BOOL IsValidStruct(HANDLE hStruct)
{
    WORD wDS;
    WORD wDSSize;
    DataStruct *pStruct;
    BOOL bReturn;

    // Get the size of the Data Segment
    // This is used to make sure that handles and pointers are
    // within the Data Segment to avoid UAEs
    _asm {
        mov ax, ds
        mov wDS, ax
    }
    wDSSize = (WORD)GlobalSize(LOWORD(GlobalHandle(wDS)));

    // Check that the handle is in the DS
    if (wDS < hStruct)
        return FALSE;

    // Lock the handle
    pStruct = (DataStruct *)LocalLock(hStruct);

    // Check that the pointer + (the size of the structure) is in
    // the DS and that pStruct is not NULL
    if (wDS < (WORD)pStruct + sizeof(DataStruct) || !pStruct)
    {
        LocalUnlock(hStruct);
        return FALSE;
    }

    // Check the verification number
    if (pStruct->nVerify == VERIFY)
        bReturn = TRUE;
    else
        bReturn = FALSE;

    LocalUnlock(hStruct);

    return bReturn;
}

```

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmLocalmem

INF: Allocation Limit on WINMEM32 Global32Alloc() Function
Article ID: Q73677

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

WINMEM32 is a dynamic-link library (DLL) that is designed to support the 32-bit flat memory model under Windows enhanced mode.

The largest block of memory that Global32Alloc() can request is (16 megabytes -- 64K). Global32Alloc allocates memory through the Windows kernel, which imposes this particular size limitation.

Adding the ability to process larger allocations is under consideration for a future version of Windows.

Additional reference words: 3.0

KBCategory:

KBSubcategory: KrMmGlobalmem

PRB: Segment Was Discardable Under 3.0 Notification
Article ID: Q81546

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

The debugging version of Microsoft Windows version 3.1 writes the following notification to the debugging terminal:

Segment was discardable under 3.0

CAUSE

One or more code segments of a dynamic-link library (DLL) are marked MOVEABLE and are not marked DISCARDABLE.

RESOLUTION

Modify the module definition (.DEF) file for the DLL to mark all MOVEABLE code segments as DISCARDABLE.

More Information:

Under Windows version 3.0, MOVEABLE code segments in a DLL are DISCARDABLE by default. This behavior changes under Windows 3.1; segments must be marked DISCARDABLE to be discarded. The debug notification highlights the change in behavior between the two versions of Windows.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrMmMemattribs

INF: GetCodeInfo() Documented Incorrectly
Article ID: Q67650

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Pages 4-159 and 4-160 in the "Microsoft Windows Software Development Kit Reference Volume 1" incorrectly documents the GetCodeInfo() function. Below is the corrected documentation for this function.

More Information:

GetCodeInfo [3.0]

Syntax void GetCodeInfo(lpProc, lpSegInfo)

This function retrieves a pointer to an array of 16-bit values containing information about the code segment that contains the function pointed to by the lpProc parameter.

Parameter Type/Description

lpProc FARPROC Is the address of the function in the segment for which information is to be retrieved. Instead of a segment:offset address, this value can also be in the form of a module handle and segment number. The GetModuleHandle function returns the handle of a named module.

lpSegInfo LPVOID Points to an array of eight 16-bit values that will be filled with information about the code segment. See the following 'Comments' section for a description of the values in this array.

Return Value None.

Comments The lpSegInfo parameter points to an array of eight 16-bit values that contains such information as the location, size, and handle of the segment and its attributes. The following list describes each of these values:

Offset Description

0 Specifies the logical-sector offset (in bytes) to the contents of the segment data, relative to the beginning of the file. Zero means no file data is available.

1 Specifies the length of the segment in the file (in bytes). Zero means 64K.

2 Contains flags which specify attributes of the segment. The following list describes these flags:

Bit	Meaning
---	-----
0-2	Specifies the segment type. If bit 0 is set to 1, the segment is a data segment. Otherwise, the segment is a code segment.
3	Specifies whether segment data is iterated. When this bit set to 1, the segment data is iterated.
4	Specifies whether the segment is movable or fixed. When this bit is set to 1, the segment is movable. Otherwise, it is fixed.
5-6	Is not returned.
7	Specifies whether the segment is a read-only data segment or an execute-only code segment. If this bit is set to 1 and the segment is a code segment, the segment is an execute-only segment. If this bit is set to zero and the segment is a data segment, it is a read-only segment.
8	Specifies whether the segment has associated relocation information. If this bit is set to 1, the segment has relocation information. Otherwise, the segment does not have relocation information.
9	Specifies whether the segment has debugging information. If this bit is set to 1, the segment has debugging information. Otherwise, the segment does not have debugging information.
10-15	Is not returned.
3	Specifies the total amount of memory allocated for the segment. This amount may exceed the actual size of the segment. Zero means 65,536.
4	Contains the handle of the segment. Zero means the segment is not loaded.
5	Contains the shift count to turn logical sector into byte offset in the file.
6-7	Reserved.

KBCategory:

KBSubcategory: KrMmMemattribs

INF: Implementing Linked Lists with Handles in Windows
Article ID: Q25064

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The standard method when using C in a non-Windows environment is to use pointers to reference the next link in the chain. Using pointers in the Windows environment requires that the `LMEM_FIXED` flag be used with `LocalAlloc()` so heap compaction will not affect the internal pointer references. Problems will occur using this technique as members of the chain are added and deleted.

More Information:

Implementing a heap compaction to free space will be difficult (or impossible). Therefore, handles should be used instead of pointers to reference successive list members in the Windows environment. Handles will allow the allocated heap data to be moved; no `LMEM_FIXED` is needed. When data in the chain must be referenced, `LocalLock()` is called using the member's handle. To traverse the linked list, do the following:

1. Given an initial handle to the chain link, call `LocalLock()`.
2. Use the address returned by `LocalLock()` to access the link's data and get the handle of the next link in the chain.
3. Call `LocalUnlock()` for the current link.
4. If not at the end of the chain, go to step 1 using a new handle.

Additional reference words: 2.00 3.00

KBCategory:

KBSubcategory: KrMmLocalmem

INF: Windows 3.1 Standard Mode and the VCPI

Article ID: Q82298

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

Windows 3.1 standard mode is compatible with expanded memory managers such as EMM386, 386MAX, and QEMM386. However, there are several problems with the Virtual Control Program Interface (VCPI) used by these products to provide access to extended memory in standard mode. Under most circumstances, users of machines with 386 or higher processors and with more than 2 megabytes (MB) of memory installed should use enhanced mode rather than standard mode.

More Information:

Windows 3.1 can run in standard mode when an expanded memory manager (EMM) is active. An EMM uses the paging mechanism of the 386 or higher processor to map extended memory blocks, as EMS pages, into the first megabyte of address space where no real memory is present. To do so, the EMM puts the processor into protected mode, and keeps the emulated upper memory block (UMB) and EMS page frames always available for processes designed for real mode (such as MS-DOS), installed device drivers, and terminate-and-stay-resident (TSR) programs. When one of these processes is running, the processor is in virtual-8086 (v86) mode, with paging on and the address mapping specified by the EMM.

In this configuration, the EMM services request to enter and leave protected mode using the VCPI. The EMM may also allocate extended memory. The standard mode MS-DOS Extender, DOSX, uses VCPI to switch to protected mode and to v86 mode. Therefore, in standard mode, the EMM remains active and continues to run "underneath" Windows. In enhanced mode, the EMM is deactivated for the duration of the Windows session.

Several questions have arisen regarding "VCPI support" under standard-mode Windows. If no EMM is installed, more extended memory and more total memory is available to Windows. An EMM uses a considerable amount of extended memory to store itself and its tables. The system incurs additional overhead because paging must be active at all times. Unless an EMM is absolutely necessary, possibly because an MS-DOS application will not run without EMS, Windows will probably run better without an EMM. Windows enhanced mode runs on a machine with 2 MB of RAM memory installed if no EMM is present.

Windows standard mode does not support an MS-DOS application that is a DPAPI (MS-DOS Protected Mode Interface) client. However, if the EMM also provides DPAPI services, DOSX will not interfere with these services.

The standard-mode task switcher attempts to arbitrate extended memory

use between Windows and MS-DOS applications. It performs this arbitration by hooking the XMS function dispatcher. This does not work properly if an EMM is installed, primarily because the extended memory portion of the address space in standard mode is not accessible to the MS-DOS portion of the switcher. For this reason, when an EMM is present in standard mode, users will experience the following types of problems:

- Performance degradation when the system switches between tasks because the task switcher cannot use XMS memory for swap space.
- Applications, such as AutoCad and Lotus 1-2-3, that include an MS-DOS extender will not run.

Because the standard-mode task switcher can't access Windows's extended memory, it can't use extended memory for swap space when an EMM is present. Users may experience some performance degradation switching between MS-DOS applications in this configuration. Enhanced mode is recommended to run many MS-DOS applications under Windows.

Windows standard mode does not specifically prevent VCPI applications from running. However, because the task switcher cannot effectively use extended memory provided by an EMM, an MS-DOS application that uses extended memory probably will fail to run in the standard-mode MS-DOS box. This applies equally to applications that use XMS, VCPI, and DPPI. While it may be possible to run an MS-DOS application that uses extended memory in an MS-DOS box under standard-mode Windows, there is no general solution to the problems involved.

The VCPI specification is maintained by Phar Lap Software, Inc., and Quarterdeck Office Systems. Windows 3.1 standard mode complies with version 1.10 of the VCPI specification. Because many commercial EMM products can be configured to not provide EMS, DOSX will attempt to use VCPI if the following are true:

- The INT 67h vector is not null
- A device named "EMMXXXX0" is present
- The VCPI detection call (INT 67h, AX=DE00h) succeeds

In particular, DOSX does not require the presence of a LIM 4.0 (Lotus/Intel/Microsoft expanded memory specification) driver or a LIM 3.2 page frame.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrMmDosdpmi

INF: Windows Enhanced Mode Allocation Limit 16 MB Minus 64K
Article ID: Q67999

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

In the "Microsoft Windows Software Development Kit Guide to Programming," page 16-13 incorrectly states:

The largest object that can be allocated in 386 enhanced mode is 64 megabytes.

This is an error in the documentation. The actual limit for a single object is 16,711,680 bytes [16 megabytes (MB) minus 64 kilobytes (K)].

More Information:

Huge memory blocks are limited to (16 MB - 64K) in 386 enhanced mode because of the way global arena headers are implemented.

GlobalAlloc() may incorrectly return a non-NULL handle when allocating objects larger than (16 MB - 64K). However, any attempt to access portions of the object beyond the limit will result in a general protection violation (reported in Windows as an Unrecoverable Application Error).

Note that GlobalSize() will return the true size of the memory allocated to the handle. For example, an application might request an allocation of 20 MB, but GlobalSize() will report how much memory was actually allocated, a figure less than or equal to (16 MB - 64K).

Additional reference words: 3.0 3.0a

KBCategory:

KBSubcategory: KrMmGlobalmem

INF: XMS Calls Under Windows 3.1

Article ID: Q83008

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

When enhanced mode Windows is running, the WIN386 module answers all extended memory specification (XMS) calls. The standard mode MS-DOS Extender (DOSX) answers all calls only in protected mode.

The standard mode task swapper answers memory allocation calls to arbitrate the use of extended memory between Windows and MS-DOS applications, and to facilitate task swapping.

More Information:

Windows enhanced mode provides its own XMS services, without regard to the XMS driver that was installed before Windows started up. The amount of XMS memory available in an MS-DOS window under enhanced mode is determined by the PIF (program information file) settings, and not by the amount of memory actually available in the system. The enhanced mode XMS driver returns failure for all XMS Lock Region calls, because enhanced mode XMS uses virtual memory rather than physical memory.

In standard mode, the task swapper hooks the XMS driver, allowing it to arbitrate XMS use between Windows and MS-DOS applications. Most XMS calls are passed through to the original XMS driver. However, calls that manage extended memory are affected by the PIF settings. Because it hooks the XMS driver, the standard mode task swapper can use extended memory allocated to Windows, but not currently in use, for swapping. This feature is disabled if a "limulator"
[Lotus/Intel/Microsoft (LIM) standard expanded memory system driver] is present.

The standard mode MS-DOS extender hooks XMS detection in protected mode. The INT 2Fh call that retrieves the address of the XMS driver's control function (AX=4310h) returns a protected mode address that an application can call to perform XMS control functions. This is intended for the use by application installation programs that must determine the version number of an installed XMS driver. Applications in the Windows environment are discouraged from making other XMS calls in protected mode. In particular, attempting to lock XMS memory in standard mode may result in a page fault or fatal system crash.

Additional reference words: 3.10 switcher

KBCategory:

KBSubcategory: KrMmMisc

BUG: GlobalPageLock Moves Memory Fixed by GlobalFix
Article ID: Q85329

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
-

SYMPTOMS

=====

When an application calls the GlobalPageLock() function specifying the handle to a block of memory that has been fixed in place by the GlobalFix() function, the address of the memory block can change.

STATUS

=====

Microsoft has confirmed this to be a problem in Windows version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

=====

The following steps demonstrate this problem:

1. Allocate a block of movable memory using the GlobalAlloc() function.
2. Fix the address of the memory block using the GlobalFix() function.
3. Increment the memory block's page-lock count using the GlobalPageLock() function.

The address of the memory block should not change between step 2 and step 3 above.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrMmFixlockwire

INF: Segment and Handle Limits and Protected Mode Windows
Article ID: Q68644

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Page 16-25 of the "Microsoft Windows Software Development Kit Guide to Programming" states:

Under the Windows standard mode and 386 enhanced mode memory configurations, there is a system-wide limit of 8192 global memory handles, only some of which are available to any given application.

With regard to standard mode, this statement is incorrect. The limit is 4096 global memory handles when Microsoft Windows runs in standard mode and 8192 when Windows runs in enhanced mode. Standard mode uses two LDT entries for each memory handle while enhanced mode uses one LDT entry for each memory handle.

More Information:

Each memory block has an associated handle that Windows and applications running in the Windows environment use to access the block. The Intel 80286 or higher microprocessor maintains records of each segment using a descriptor, which describes the attributes of each memory block such as its base address, size, attributes (read, write, execute), and privilege level. In enhanced mode, each memory block has one handle and at least one descriptor. In standard mode, each memory block has at least two descriptors. Because Windows and its applications run in 16-bit protected mode, global memory blocks larger than 64K (the maximum segment size in 16-bit protected mode) use several descriptors.

The architecture of the Intel CPUs requires that each descriptor table fit into its own 64K segment. Because each descriptor is 8 bytes long, each descriptor table can hold a maximum of 8192 descriptors (64K / 8 bytes = 8K). Therefore, each descriptor table can manage up to 8192 segments. For more information, see the "386 DX Microprocessor Programmer's Reference Manual," (Intel Corporation).

Windows creates a global descriptor table (GDT) for the system, and one local descriptor table (LDT) for itself and all applications running under Windows. Because all applications use the same LDT, a maximum of 8192 descriptors are available to Windows and applications. Enhanced mode can manage a maximum of 8192 segments because each segment requires one descriptor. Standard mode can manage a maximum of 4096 segments because each segment requires two descriptors. Because descriptors are used for code, data, and resource segments, the maximum number of descriptors available for applications to use for global memory blocks is considerably less than 8192 and 4096.

Windows does not provide way to determine how many selectors are available for a given application to use. All Windows API functions fail when there is not enough free memory to satisfy a given request, or when there are not enough descriptors available. Either way, Windows is unable to satisfy the application's memory request. It is important that selectors be used with deliberation because each Windows application must cooperate with other Windows applications. Instead of allocating many small blocks from the global heap, an application should allocate fewer blocks that are larger in size. The application can then divide each larger memory block into pieces for its own use.

If small memory blocks are required by the application, use the local memory-management routines provided by Windows. A local handle does not impact the selector limit at all because a local memory block is allocated inside a global memory block. For example, the LocalAlloc function allocates a memory block from an application's or a dynamic-link library's (DLL's) local heap.

In some circumstances, it may be advantageous to employ a more sophisticated memory-management scheme called multiple FAR heaps. This technique is useful if the application requires a number of blocks of memory, say 100, and the total number of bytes in each block is not above 64K. Chapter 18 (pages 707-724) of "Windows 3.0 Power Programming Techniques," by Paul Yao and Peter Norton (Bantam Computer Books), contains more details on this technique. In this chapter, Norton and Yao state the steps necessary to perform local heap allocation in a dynamically allocated segment. This chapter effectively describes the technique of using multiple FAR heaps and also provides some sample code.

This technique involves four steps:

1. Globally allocate a block of memory with GlobalAlloc.
2. Lock the block of memory with GlobalLock.
3. Initialize the block of memory by calling LocalInit.
4. Modify the local memory-management routines so each updates the DS register to point to the new heap just before a call to a local memory-management routine. When access to the new heap is complete, immediately restore the DS register.

Overall, this technique uses fewer global handles and less memory overhead with each call.

Additional reference words: 3.00 MICS3 R3.2

KBCategory:

KBSubcategory: KrMmGlobalmem

PRB: XMS Version Information in MS-DOS Window Incorrect
Article ID: Q83455

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

SYMPTOMS

In an MS-DOS window under Windows, the version number of the extended memory system (XMS) driver is incorrect. Windows exhibits the same behavior for its expanded memory system (EMS) driver, MS-DOS Protected Mode Interface (DPMI), or any other system that enhanced mode Windows virtualizes.

CAUSE

The application retrieves the version number of the XMS driver in enhanced mode Windows, not the version number of the "real" XMS driver (which was present before Windows startup).

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrMmDosdpmi

INF: Shorthand Notation for Memory Allocation Flags

Article ID: Q72459

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

Some flags have been defined in the WINDOWS.H file [distributed with the Windows Software Development Kit (SDK)] that are designed to be used instead of common flag combinations for memory allocations. Included below is an explanation of these flags:

Global Memory Allocations

There are two flags defined for global memory allocation: GHND and GPTR. Both are to be used for the wFlags parameter of the GlobalAlloc function.

Flag	Definition
----	-----
GHND	(GMEM_MOVEABLE GMEM_ZEROINIT)
GPTR	(GMEM_FIXED GMEM_ZEROINIT)

Local Memory Allocations

Similarly, there are two flags defined for local memory allocation: LHND and LPTR. Both are to be used for the wFlags parameter of the LocalAlloc function:

Flag	Definition
----	-----
LHND	(LMEM_MOVEABLE LMEM_ZEROINIT)
LPTR	(LMEM_FIXED LMEM_ZEROINIT)

The LPTR flag returns a pointer that can be used immediately (no need to call the LocalLock function); whereas with the GPTR flag, the GlobalLock function still must be used on the handle that is returned.

For more information on the GlobalAlloc and LocalAlloc functions, refer to the "Microsoft Windows Software Development Kit Reference Volume 1".

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrMmMemattribs

INF: Appropriate Uses of WINMEM32

Article ID: Q73679

The information in this article applies to:

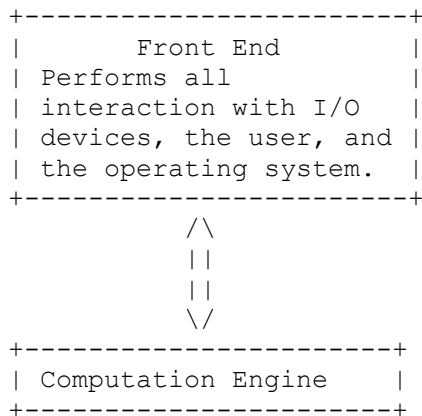
- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

WINMEM32 is a dynamic-link library (DLL) that supports 32-bit flat memory under the Microsoft Windows graphical environment. WINMEM32 provides a flat model computing environment for applications ported from another operating system to Windows. New applications should not be designed using WINMEM32.

More Information:

WINMEM32 is designed for an application structured as follows:



The front end application that provides the user interface is a standard application for the 16-bit Windows environment. WINMEM32 supports the 32-bit "computation engine" that has no user interface and does not interact with any peripheral devices or the operating system.

Windows is not designed to support 32-bit applications; WINMEM32 is a temporary measure to support an independent software vendors (ISV) with a substantial investment in 32-bit code developed for another operating system. Even with WINMEM32, porting an application to Windows involves many complex issues that an ISV must carefully address for the application to work properly.

Future versions of Windows will address the 32-bit application issues better than WINMEM32 does. Future versions of Windows will support WINMEM32 applications even though WINMEM32 is not developed any further.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrMmWinmem32

INF: What EMS Means to Developers

Article ID: Q30590

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

This article describes how using Windows expanded memory affects application development.

Please note that when expanded memory is mentioned in this article, it also means extended memory. Extended memory is used as expanded memory by Windows.

More Information:

In Windows versions 2.x and 3.0, expanded memory can be used by the Windows memory manager. In enhanced mode Windows 3.0, the memory manager can emulate expanded memory by using extended memory. All Windows applications are run in one virtual machine, but each DOS application receives its own virtual machine in which to run.

In real mode Windows, expanded memory presents the following problems:

1. Applications cannot share memory through global handles
2. The efficient use of available EMS (expanded memory specification) in a small frame

The only way applications can share global data is through the dynamic data exchange (DDE) protocol or through the clipboard. When large frame EMS is used, global objects are allocated above the EMS line and are available only to one application. For example, in large frame EMS, task A allocates some global memory and passes the handle to task B. When task B attempts to access the memory object, it will not be in memory. Instead, task B will access the corresponding location in its own EMS memory.

DDE and clipboard objects can be shared because the Windows memory manager will copy the contents of these objects to the currently running application's EMS bank. The application must check the return value from GlobalLock() because the memory manager may not be able to successfully copy the object in a low memory situation. Clipboard objects should be copied by the receiving application before the clipboard is closed because the memory manager may delete the global object.

Applications can share data through dynamic-link libraries (DLLs), as well as through DDE and clipboard objects; however, only data located in the DLL data segment can be shared. Global objects allocated by DLLs are allocated above the EMS line, just like objects allocated by an application.

There is one exception to the rule of not sharing global memory handles. If the `GMEM_NOT_BANKED` flag is used, the object will be allocated below the EMS bank line, making it available to all applications. This should not be done because there is only a limited amount of memory below the EMS line when the system is using large frame EMS. This flag is available for device drivers that require this kind of shared memory; applications should be designed using one of the other methods to share memory objects.

The second issue regards the most effective use of small frame EMS by applications. In small frame EMS, code and resources are the only EMS items available to applications. Once code and resources are loaded into EMS, they will not be discarded. However, they are banked out when another task is running.

The Windows loader fills EMS before it loads code or resources into conventional memory. For this reason, the DEF file for each application should list the segments that are important throughout the life of an application at the top of the list of segments in the SEGMENTS statement, and these segments should be marked as PRELOAD. List the segment that contains the application's main window procedure first. The segments for code that is used and discarded, such as initialization code, should be listed toward the bottom of the list. By the same reasoning, mark resources that are important throughout the life of the application as PRELOAD.

For small applications, the `LimitEmsPages()` function allows an application to limit the amount of EMS that is allocated on its behalf. The argument passed to this function is the maximum number of kilobytes of EMS that Windows will allocate for the application. A small application can call this function to keep 400-500K of EMS from being allocated when the application can run reliably with 100K of EMS.

When `LimitEmsPages()` is used, it is important to note that in large frame EMS (where DLL code is located above the EMS line), a small application will require more space than the sum of its segments because library code shares the EMS allocated for the application.

Note: Hook callback functions must be placed in FIXED DLL code so the code will always be present regardless of EMS state. The same is true for `GlobalNotify()` callback functions.

Additional reference words: 2.03 2.10 3.00

KBCategory:

KBSubcategory: KrMmMisc

FIX: GlobalReAlloc() Shrinks >1 MB Block to <1 MB UAE

Article ID: Q70819

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9103002

SYMPTOMS

When the GlobalReAlloc function is used to change the size of a memory block larger than 1 MB to smaller than 1 MB, an unrecoverable application error (UAE) occurs either immediately or when the handle is freed by the GlobalFree function.

CAUSE

In this situation, the GlobalReAlloc function corrupts pointers in the huge memory object.

RESOLUTION

Microsoft has confirmed this to be a problem in Microsoft Windows version 3.0. To avoid this problem, use the GlobalAlloc function to allocate a new block of memory. If required, copy a portion of the contents of the original block to the new block. Free the original block with the GlobalFree function.

This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmGlobalmem

INF: Corrected WINMEM32.DLL Available in Software Library
Article ID: Q71752

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

An update to the WINMEM32.DLL file included with the Windows Software Development Kit (SDK) version 3.0 has been placed in the Software/Data Library. This update corrects a problem related to freeing memory allocated by the Global32Alloc function. With the original version of this file, the Global32Free function can fail to free an allocated object.

Applications developed to use WINMEM32.DLL should include this updated version of the library. Because WINMEM32.DLL is not included with the retail Windows product, this does not place any additional requirements on the software developer.

The updated WINMEM32.DLL has been compressed and stored in the Software/Data Library in a file named WINMEM32. WINMEM32 can be found in the Software/Data Library by searching on the keyword WINMEM32, the Q number of this article, or S13019. WINMEM32 was archived using the PKware file-compression utility.

The version of the WINMEM32.DLL file shipped with the Windows SDK version 3.1 corrects this problem with the Global32Free function.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmWinmem32

INF: Maximizing the Use of Available Memory in Windows
Article ID: Q72236

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The Microsoft Windows graphical environment creates and stores objects on behalf of each application in the system. Two places store many of these objects, the user heap and the graphics device interface (GDI) heap, each one limited to 64K. This article discusses the objects, their size, and how to maximize the use of the heaps.

More Information:

A good way to see what is stored in the heaps is to use the Heap Walker tool (HEAPWALK.EXE) provided with the Microsoft Windows Software Development Kit (SDK). Heap Walker is documented in Chapter 11 of the SDK Tools manual for Windows 3.0 and in Chapter 9 of the SDK Programming Tools manual for Windows 3.1. The memory management practices of Windows are documented in Chapters 15 and 16 of the SDK Guide to Programming for Windows 3.0. Further information on Windows memory management is available in Charles Petzold's "Programming Windows" (Microsoft Press) and in Peter Norton and Paul Yao's "Windows 3.0 Power Programming Techniques" (Bantam Computer Books).

The following table lists the objects stored in the user heap and the typical sizes for these items:

Object	Size in Bytes
-----	-----
Menu	20 + 20 per menu item
Window Class	40 to 50
Window	60 to 70

Note that every running program requires space in the user heap. Every application must use this shared resource wisely. One technique to reduce heap requirements is through the judicious use of resources. For example, static strings should be placed into a string table instead of being stored as string variables. If a group of applications shares a common set of resources, place the resources into a dynamic-link library (DLL). Multiple applications can share one copy of code, data, and resources through a DLL.

Another way to reduce heap requirements is through the use of class extra bytes and window extra bytes. Although these bytes are stored on the user heap, each is associated with a particular window or window class. These bytes are convenient places to store a handle to a data structure that has been allocated from global memory.

Menus are, by far, the biggest consumer of user heap space. Applications that have multiple menu bars or create menus with the

TrackPopupMenu function should load these resources only as needed and destroy them after use, instead of waiting for program termination. In Windows 3.1, user stores menus in a separate 64K heap.

When an application creates a GDI object, Windows allocates space from the GDI heap. While most applications create GDI objects, an application should not create too many objects at one time. Also, each object must be destroyed when it is no longer required by the application. The following table lists the objects stored in the GDI heap and their typical sizes:

Object	Size in Bytes
-----	-----
Brush	32
Bitmap	28 to 32
Device Context (DC)	300
Font	40 to 44
Pen	28
Region	28
Palette	28

Items are created in the GDI heap whenever an application creates a GDI object. Most applications create GDI objects, but an application should not create too many objects at one time. Also, an object must be destroyed when it is no longer required by the application.

The maximum number of windows that can be open simultaneously is constrained by the amount of space remaining in the user heap. As noted above, the user and GDI heaps are each limited to 64K.

Because heap space is shared among all running applications, an application must check the value returned from each function call to verify that memory allocations are successful. The debugging version of Windows produces a FatalExit message when an application uses an invalid handle. This information is difficult to obtain from any other source.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmGlobalmem

INF: Checksums for Windows Executable Image Files

Article ID: Q72471

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

When the debugging version of Windows is running in real mode, it calculates the checksum of each code segment that is loaded or unloaded. This calculation is performed to ensure that a code segment is not modified by a "wild write" (storing data through an invalid pointer).

An application can cause the debugging version of Windows to calculate checksums on all segments present in the system by calling the `ValidateCodeSegments` function.

`ValidateCodeSegments` can be overridden by adding an `EnableSegmentChecksum=0` line to the `[kernel]` segment of the `WIN.INI` file. However, when Windows 3.0 is running in real mode with EMS (expanded memory specification) memory, the kernel does not calculate checksums.

Segment validation is performed only in real mode because applications are prevented from writing into code segments in protected mode. Any attempt to write into a code segment in protected mode causes an unrecoverable application error (GP-fault), which provides immediate notification of the programming error.

Additional reference words: 3.00

KBCategory:

KBSubcategory: `KrMmMisc`

FIX: Microsoft Windows Page Locks GMEM_FIXED Memory

Article ID: Q72584

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

When an application running under Microsoft Windows version 3.0 allocates a block of memory with the GMEM_FIXED attribute specified, Windows returns the handle to a block of page-locked memory. Depending on the amount of available memory, the system may hang or slow down because of disk thrashing.

CAUSE

The Windows kernel page locks all fixed memory allocated in Windows version 3.0. When a memory block is page locked, Windows cannot page to disk the physical memory associated with the segment.

This behavior is incorrect for applications; however, it is correct for dynamic-link libraries (DLLs). If a DLL allocates GMEM_FIXED memory, Windows provides page-locked memory because Windows assumes that the DLL might access the memory at interrupt time.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmMemattribs

PRWIN9106004: Memory Allocation in Enhanced Mode Hang or UAE
Article ID: Q73336

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9106004

SYMPTOMS

Under certain circumstances, when an application calls the GlobalAlloc or GlobalReAlloc function, enhanced mode Windows 3.0 hangs or produces an unrecoverable application error (UAE).

CAUSE

The enhanced mode of Windows version 3.0 improperly handles heap partition marker arenas. Encountering one should signal the end of a given free-space search.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.00a.

Additional reference words: 3.00 3.00a

KBCategory:

KBSubcategory: KrMmGlobalmem

INF: Information About Headings and Labels in HEAPWALK

Article ID: Q74301

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The Heap Walker utility (HEAPWALK) is a tool used to view the memory layout in the Microsoft Windows graphical environment. With HEAPWALK, you can examine memory before, during, and after an application runs and examine how an application behaves under different memory conditions. This article lists and explains the columns of information that HEAPWALK displays for global and local memory objects.

The version of HEAPWALK provided with version 3.1 of the Microsoft Windows Software Development Kit (SDK) differs significantly from the version provided with version 3.0. HEAPWALK version 3.1 uses the Tool Helper dynamic-link library (DLL), displays information in a more organized manner (by sorting objects based on a secondary key), provides information about objects in the user heap as well as in the GDI heap, and simultaneously displays a graphical representation of a resource along with a hex dump of the resource.

The information in this article applies to HEAPWALK versions 3.0 and 3.1 unless otherwise noted.

More Information:

Main Window

HEAPWALK displays a list of memory objects in the global heap in its main window. Options on the Walk menu determine which objects HEAPWALK displays in its main window. Options on the Sort menu determine the order in which objects are displayed in the main window. Options on the Object menu specify a number of actions that HEAPWALK can perform on a selected object.

For each global memory object, HEAPWALK displays various information in its main window, as follows:

ADDRESS:

Address of the memory block (in hexadecimal).

- In real mode or standard mode, this is a physical address value.
- In enhanced mode, this is a linear address value. (The paging mechanism maps a linear address to a physical address.)

HANDLE:

Handle for the memory block (in hexadecimal).

- This value was returned from a GlobalAlloc call.

SIZE:

Size of the memory block in bytes (in decimal).

LOCK:

Current lock count of the memory block (displayed only if nonzero).

FLG:

Flag.

D - If memory block is discardable.

Page 11-3 of the "Microsoft Windows Software Development Kit Tools" manual for version 3.0 incorrectly states that this field contains an "S" for shareable segments. Instead, the OBJ-TYPE field contains "Shared" for shareable segments.

OWNER-NAME:

Owner of the memory block.

- Usually the module that allocated the memory block.
- Can be an application name (for example, HEAPWALK), one of the Windows modules (USER, GDI, DISPLAY, and so on), or FREE.

OBJ-TYPE:

Type of memory object:

- Code segment number or name - Code segment along with its number or name. (To display segment names, place the SYM file for the application module in the directory from which HEAPWALK runs, and select the Label Segments command from the Sort menu.)
- Data - Data segment.
- Shared - A shared object.
- Resource
- Kernel data structures such as:
 - Task
 - Task Database
 - Module Database

ADD-INFO:

Additional information that may accompany OBJ-TYPE.

- This is usually a resource type such as an icon, a cursor, or a font.

Local Window

To view the memory objects in an application's local heap, select the application's data segment and use the Show and LocalWalk commands from the Object menu. This results in two windows: a Show Window, which displays a hexadecimal memory dump, and a Local Window, which displays a list of the local objects and their attributes, as follows:

OFFSET:

Offset of the local memory block in the data segment (in hexadecimal).

SIZE:

Size of the local object in bytes (in decimal).

MOV/FIX:

Mov - For a movable memory block.

Fix - For a fixed memory block.

FLAGS:

For movable objects.

D - If discardable.

Lnn - Lock count if nonzero.

OBJECT TYPE:

"Free" for free objects.

For the GDI data segment, the local window displays the object type
(for example, DC, Brush, Pen, or Region).

Show Window

Using the Show command from the Object menu displays a hexadecimal dump of any global memory block in a separate window. The show window displays the offset, hexadecimal memory dump, and the ASCII memory dump. The Showbits command displays the memory as a bitmap (if the block contains a cursor, an icon, or a bitmap).

HEAPWALK version 3.0 truncates objects that are larger than 64K.

HEAPWALK version 3.1 does not exhibit this behavior.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrMmMemattribs

INF: Future Direction of WINMEM32

Article ID: Q74478

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

WINMEM32 is a dynamic-link library (DLL) that allows applications running in enhanced mode Windows version 3.00 to support the 32-bit flat memory model.

As additional versions of the product are released, it is planned that:

- The limit on the maximum object size will be increased from 16,711,680 bytes (16 megabytes minus 64 kilobytes).
- Selectors for WINMEM32 memory objects will not be tiled by default. Instead, selectors will be tiled using the Global16PointerAlloc() function. This change will make WINMEM32 less "selector intensive."

In the future, a 32-bit version of Windows will be released. At that time, all development of WINMEM32 will cease because its functionality (supporting 32-bit applications on 16-bit Windows) will be unnecessary. WINMEM32 applications will be supported on this future system.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmWinmem32

INF: Using Memory Below 1 Megabyte

Article ID: Q74696

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

Memory allocated with a base address below 1 megabyte (low memory) is useful for communicating between applications developed for the Microsoft Windows graphical environment and MS-DOS terminate-and-stay-resident (TSR) programs and device drivers.

The only way a Windows application can directly allocate memory guaranteed to be below 1 MB is to use the GlobalDosAlloc function. However, this memory is a limited resource and should be used with care.

More Information:

The following clients use memory below 1 MB:

- MS-DOS
- MS-DOS device drivers
- TSRs
- Parts of the Windows kernel
- Windows enhanced mode virtual drivers (VxDs)
- Windows applications that call GlobalDosAlloc
- Windows applications that call GlobalAlloc and receive low memory by chance
- The task database for each active Windows application (This small block of low memory holds data used by MS-DOS.)

The first four clients on the list allocate their memory before any Windows applications are run, therefore an application cannot prevent this consumption of the low memory resource. (The user can modify the CONFIG.SYS and AUTOEXEC.BAT files to reduce the number of devices and TSRs.)

Virtual devices can allocate (or map) memory below 1 MB to communicate with various hardware devices and MS-DOS device drivers. VxDs such as the virtual NetBIOS driver and various virtual display drivers map memory below the 1 MB line, reducing the amount of low memory available to Windows.

The last three clients of low memory listed can progressively consume more of the resource as the system runs, therefore an application can increase the chance that its low memory allocations will succeed by performing them as early as possible during system initialization. Two methods for doing this are:

1. Load the application or dynamic-link library (DLL) from the "run=" or "load=" line in the WIN.INI file or from the StartUp group

provided by Windows 3.1, or

2. Create a Windows device driver that performs the allocation when it is first loaded.

Beware of using too much low memory because other applications that need low memory may begin to fail. The worst outcome of allocating too much low memory is that Windows will be unable to allocate the task database required to start an additional application.

In Windows enhanced mode, the lower memory that Windows applications allocate is local to the system virtual machine (VM). Other virtual machines (or MS-DOS compatibility boxes) cannot see the memory that the GlobalDosAlloc function allocates. Allocating "global" low memory (seen by all virtual machines) requires a virtual device, or the memory must be allocated before Windows is loaded.

Additional reference words: 3.00 3.10 meg

KBCategory:

KBSubcategory: KrMmDosdpmi

Sample: Global Heap Functions

Article ID: Q97940

Summary:

SHOWMEM is a sample application that allows the user to allocate, free, and manipulate memory from the global heap. Different "what if" scenarios can be demonstrated without writing and rewriting code. Information about global heap objects is displayed using the GlobalEntryHandle and ModuleFindHandle TOOLHELP.DLL functions. Additional information about how to use the global heap functions is provided, as well as a collection of related Microsoft Knowledge Base articles in help-file format.

SHOWMEM can be found in the Software/Data Library by searching on the the word SHOWMEM, the Q number of this article, or S14158. SHOWMEM was archived using the PKware file-compression utility.

Additional reference words: 3.10 tool help helper toolhelper

KBCategory:

KBSubcategory: KrMMGlobalMem

INF: DPMI Specification Available from Intel

Article ID: Q62065

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The MS-DOS protected mode interface (DPMI) specification is available free of charge by calling Intel Corporation at (800) 548-4725. International customers can obtain the DPMI Specification by contacting the Intel sales office that serves their country.

More Information:

The DPMI specification was defined to allow an MS-DOS program to access the extended memory provided by a PC architecture computer while maintaining system protection. DPMI defines a specific subset of MS-DOS and BIOS calls that can be made by protected mode MS-DOS programs. It also defines a new interface through software Interrupt 31h, which protected mode programs can use to allocate memory, modify selectors, call real mode software, and so forth.

DPMI is commonly used to communicate with a terminate-and-stay-resident (TSR) program or an MS-DOS device driver from a protected mode application. If the protected mode application passes a buffer of data to a TSR or device driver, the application must allocate the buffer from memory addressed below 1 megabyte to make the data accessible to the real mode software. The application must also translate the buffer's address from a selector address to a segment address. If the real mode software calls back to a function in the protected mode application, the application must allocate a real mode callback address. DPMI provides services to perform each of these tasks.

The Microsoft Windows standard mode MS-DOS extender and Windows enhanced mode provide translation services for most of the commonly used interrupts. This allows a driver or an application to call MS-DOS, the BIOS, and other common services without using the DPMI. However, when an application communicates with a network, a TSR, or any real mode software for which Windows does not provide automatic translation, it must use DPMI services.

DPMI services should be used only in a Windows device driver or a dynamic-link library (DLL). An application should manipulate selectors using the kernel selector functions, documented in the Microsoft Windows Software Development Kit (SDK). Calling DPMI services from an application may not be supported by future versions of Windows. However, calling these services from a device driver or a DLL will be supported.

Additional reference words: 3.00 vcpi

KBCategory:

KBSubcategory: KrMmDosdpmi

INF: Using GlobalNotify to Implement Real Mode Virtual Memory
Article ID: Q74796

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

In the Microsoft Windows graphical environment, the GlobalNotify function works with memory allocated using the GMEM_NOTIFY and GMEM_DISCARDABLE flags as follows:

1. The memory is allocated.
2. Windows runs low on memory, so it discards the least-recently used (LRU) block of discardable memory.
3. If the block to be discarded was allocated with the GMEM_NOTIFY flag set, Windows calls the appropriate notification procedure.
4. The notification procedure gives permission to discard the memory.

This article discusses these four steps and some related issues.

More Information:

The notification procedure must be in a fixed code dynamic-link library (DLL). This ensures that the memory containing the notification procedure will not be discarded. Although the application allocated the memory, the application (except for the notification procedure) is not in memory because memory is low.

The application has two methods available to instruct the notification procedure regarding a particular block of memory:

1. Store disposition instructions at the beginning of the block of memory. For example, if the notification should save 1000 bytes of the block in the TEMP.VM file when the block is discarded, reserve a fixed number of bytes at the beginning of the memory block for a header. The header could contain the name of the file (TEMP.VM) and the number of bytes to store.
2. Assign the job of allocating the memory to the DLL. Then, when the notification routine is called, the DLL has the data on where to save the contents of the memory block. This can be accomplished by implementing the MyGlobalAlloc and MyGlobalLock functions in the DLL.

MyGlobalAlloc allocates an appropriately sized block of discardable memory with the notify bit set, and stores the handle returned into a handle table (stored in the DLL's data segment). MyGlobalLock locks the appropriate block of memory and returns a pointer to the memory to the calling application.

The following issues must be addressed in the notification routine:

1. Open the file, save the data, and close the file upon receiving the notification. Applications should not leave file handles open between messages. Using a file handle that is already open can cause disastrous results because the current program segment prefix (PSP) may not belong to the application, and the application might use the wrong file handle.
2. Do not call any functions that might move memory, such as `GlobalAlloc`, `GlobalReAlloc`, or `LocalAlloc`, in the notification function. If the notification procedure calls the `SendMessage` function, be sure it does not start a chain reaction that could move memory. Using `PostMessage` is a safer alternative.
3. The `GlobalLock` function fails if the memory has been discarded; therefore, the DLL should have a `MyGlobalLock` function that follows this algorithm:

```
if (GMEM_DISCARDED & GlobalFlags(hMem))
{
    hMem = GlobalReAlloc(hMem, lSize,
                        GMEM_DISCARDABLE | GMEM_NOTIFY | GMEM_MOVEABLE);

    // Reload the data here.
}

lpMem = GlobalLock(hMem);
```

A typical notification routine in the DLL might resemble the following:

```
BOOL FAR PASCAL NotifyProc(HANDLE hMemToDiscard)
{
    BOOL bValid;

    bValid = CheckForKnownMemory(hMemToDiscard);

    return bValid;
}
```

The `CheckForKnownMemory` function compares the handle specified with the function's list of known handles. If the function recognizes the handle, the function saves the contents of the memory block to disk and returns `TRUE`; otherwise, the function returns `FALSE`.

This function is initialized by adding the following line to the application's `WinMain` function:

```
GlobalNotify(NotifyProc);
```

Calling the `MakeProcInstance` function to create a procedure instance address is not necessary because the function is in a DLL. The `NotifyProc` function must be in a fixed code DLL and must be exported. The `CheckForKnownMemory` function should be in the same

code segment as NotifyProc for minimal movement of memory.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmGlobalnotify

INF: Solving the
Article ID: Q105274

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
-

SUMMARY

=====

Each task under Windows requires a data structure allocated below 1 megabyte (MB) called the Task Data Base (TDB). Under the debug version of Windows, failure of a TDB's allocation shows up as:

```
t Kernel: GlobalAlloc(200) failed for ????
```

This error occurs within the context of LoadModule as it attempts to start a new task under a low conventional memory condition.

The most common cause of low conventional memory is fixed allocations made on behalf of a dynamic-link library (DLL). Fixed allocations should be used only for code and data touched at interrupt time. All other allocations should be made with the moveable attribute.

MORE INFORMATION

=====

The Windows heap consists of all conventional and XMS memory available after WIN.COM and WIN386.EXE/DOSX.EXE are loaded by MS-DOS. These two physically separate blocks are combined into a single linear address space to make up the global heap.

Windows allocates fixed objects with a bottom up/first fit algorithm. As more fixed objects are allocated from the heap, conventional memory gets pinched. Eventually, even a small allocation for the TDB fails resulting in LoadModule returning 0 (zero).

The HEAPWALK.EXE utility provides a view onto the global heap to determine what is using up conventional memory. The Sort Address menu item sorts the main heap with lower addresses at the top of the list. Any object with an address of 9FFFF or below is in conventional memory. An object with an "F" set in the FLG column is fixed. Any fixed object owned by an application or its dependent dynamic-link libraries that is not a TDB (TYPE Task in HEAPWALK) should be considered suspect.

The module definition (.DEF) file should mark CODE MOVEABLE DISCARDABLE and DATA MOVEABLE. Any allocations should contain the GMEM_MOVEABLE attribute. The EXEHDR.EXE utility can be used to identify modules that use the fixed attribute for their code or data segments. Segments without the <moveable> attribute in the "flags" column of EXEHDR's output are fixed.

Additional reference words: 3.10 dll tdb memory low conventional fixed
KBCategory:
KSubcategory: KrMmInsuffmem

INF: Determining Free Memory in Windows Enhanced Mode
Article ID: Q74941

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

When an application requires memory for a particular purpose, it should request that amount of memory from the system. If a given request fails, the application can present an error message to the user, or make a smaller request.

In the MS-DOS (non-Windows) environment, it is customary for an application to request that the operating system determine how much memory is free and report that information. The application can then allocate that amount of memory and scale its capacity limits accordingly. This is acceptable in an environment where only one application is running at any given time, which has complete access to all system resources available.

However, in the Windows cooperative multitasking environment, applications must share system resources with other applications running simultaneously.

More Information:

In enhanced-mode Windows, determining the amount of free system memory is a very complex problem because Windows uses virtual memory. There are also a number of different types of memory that are used for specific purposes. The following list enumerates some of these memory options:

- DDE share
- Discardable
- Fixed
- Movable
- Page locked
- Pageable
- Provided by GlobalDOSAlloc

The presence of any of these attributes will affect the amount of free memory.

It has been suggested that to attempt an allocation and then properly handle failure by potentially trying another allocation is too slow. However, it is doubtful that any method of calculating available memory will be any faster (if such a calculation was even possible). An overriding complication is that any memory use in another application or in a virtual machine will invalidate any computed value.

An excellent discussion about dealing with varying amounts of system memory is in Chapter 18 of "Peter Norton's Windows 3 Power Programming Techniques" (Bantam Books, 1990) beginning on page 661. Given the caveats above, it is possible to obtain a very rough estimate of free system memory. Two functions report this information: the GlobalCompact function and the MS-DOS Protected Mode Interface (DPMI) function 0500h (get free memory information).

There are two pools of memory in enhanced-mode Windows:

1. The DPMI memory pool managed by the WIN386 paging memory manager. Use DPMI function 0500h to determine the size of this pool.
2. The global heap memory pool(s) managed by KRNL386.EXE (the Windows enhanced-mode Kernel. Use the GlobalCompact(-1) function to determine the size of this pool.

A rough estimate of available memory can be computed by placing the following code fragment into an application:

```
FreeMemEst = max(GlobalCompact(-1),  
                 (DPMI_Call_AX_0500h->MaxUnlockedPageAlloc - 1L) * 4096L);  
FreeMemEst = min(FreeMemEst, (16L * 1024L * 1024L) - (64L * 1024L));
```

The first line of code determines available memory according to both memory managers. The second line accounts for the limitation imposed by the GlobalAlloc function, which sets the maximum size of a segment at (16 megabytes minus 64 kilobytes). The GlobalCompact function does require a lot of processing time, especially in standard (286 protect) mode.

Additional reference words: 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrMmMisc

INF: Demand Paging MS-DOS Applications

Article ID: Q47125

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The following article has two parts. The first part applies to Windows version 3.0, and to Windows version 3.1 without 32-bit disk access. The second part applies to Windows version 3.1 with 32-bit disk access.

More Information:

Part 1: Windows 3.0, and Windows 3.1 Without 32-Bit Disk Access

Under enhanced mode Windows version 3.0, it is not possible to allow MS-DOS applications to use virtual (demand paged) memory (VM) when they are active (they are the foreground application, or have the background execution option set). This is because the paging mechanisms of the system use either:

- MS-DOS plus an MS-DOS device plus INT13
- Just INT13

In other words, to access the paging file (virtual memory that is currently demand-paged out) Windows must call code that resides in the VM along with the application (BIOS, MS-DOS, and so on).

The problem this causes is best illustrated by an example:

1. Run an MS-DOS application that hooks INT 13h.
2. Arrange to "page out" the page that has part of the application's INT 13h hook in it.
3. Touch a "not present" page.
 - a. This generates a page fault.
 - b. The system calls "page this page in."
 - c. This calls INT 13h.
 - d. This page faults because the INT 13h hook of the application is currently paged out.
 - e. The system calls "page this page in."
 - f. This calls INT 13h.
 - g. This page faults because the INT 13h hook of the application is currently paged out.

This continues forever, causing a deadlock situation, and the system halts.

To prevent this problem, make sure every page of an MS-DOS application is always present when the MS-DOS application's VM is active. Thus, MS-DOS application VMs are always present, and therefore, do not use demand-paged memory except when they are not active.

Given the way that system paging works, there is no way to work around this limitation, partly because the installed MS-DOS application base is so diverse. Note also that this problem with INT 13 is only part of the picture. Almost all hardware interrupts can cause exactly the same problem.

Part 2: Windows version 3.1 with 32-Bit Disk Assess

One solution is to change the way system paging works so that to access the paging file, the system paging does not have to use any code that resides in the VM with the application. 32-bit disk access for Windows version 3.1 performs this type of functionality.

32-bit disk access addresses the problem that prevents the demand paging of MS-DOS application VMs. 32-bit disk access allows the paging component of the system to access the paging file without having to access anything that resides in the VM (the application, MS-DOS, the BIOS, and so on). This prevents the paging deadlock problem discussed above because all of the code and data needed to access the paging file is inside enhanced mode Windows and is designed such that access to the paging file will never cause a page fault to occur.

As a side benefit, this is also much faster than calling MS-DOS or some other component in the VM because calling VM code requires that the 80386 CPU change modes between protected mode and virtual mode. The transitions between CPU modes are quite expensive in terms of CPU cycles. Therefore, the other benefit of 32-bit disk access is that there is an improvement in "paging throughput" because the "CPU mode transitions" required by access to the paging file through MS-DOS or INT 13h are eliminated.

Additional reference words: 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrMmMisc

PRB: GlobalUnlock Can Cause Fatal Exit 0x02F0

Article ID: Q49838

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

Under the debugging version of Windows, when an application calls the GlobalUnlock function, a FatalExit 0x02F0 "GlobalUnlock: Object usage count underflow" error occurs.

CAUSE

The application called the GlobalUnlock function more times than it called the GlobalLock function. Under the retail version of Windows, the function returns the normal value 0 in these circumstances.

RESOLUTION

Match each GlobalLock call with a GlobalUnlock call.

Additional reference words: 2.03 2.10 3.00 3.10 2.x SR# G891012-113
fatal exit 2F0 02F0 2F0h

KBCategory:

KBSubcategory: KrMmFixlockwire

PRB: Reset A20 Bit Set During DPMI Simulate Interrupt Crash
Article ID: Q76582

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

A Windows application that specifies "reset interrupt controller and A20 line" when calling the MS-DOS protected mode interrupt (DPMI) function "Simulate Real Mode Interrupt" can cause Windows to crash.

RESOLUTION

Applications that use the "Simulate Real Mode Interrupt" function must ensure that this bit is not set.

More Information:

The "Simulate Real Mode Interrupt" function is documented in the DPMI specification.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmDosdpmi

PRB: Protected-Mode GlobalCompact Return Is Not Free Memory
Article ID: Q76686

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

When Windows is running in one of its protected modes, the value returned by the GlobalCompact function does not accurately reflect the amount of free memory available in the system.

In enhanced mode, Windows can swap memory objects to disk. This process takes a long time relative to accessing an object in memory. Therefore, the GlobalCompact function returns the amount of memory available without performing any paging.

In standard mode, the GlobalCompact function never reports more than 1 megabyte (MB) of memory free because of a memory allocation limit on the 80286 chip.

STATUS

Microsoft has confirmed that this problem occurs in Windows versions 3.0 and 3.1. We are researching this problem and will post new information here as it becomes available.

Additional reference words: 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrMmGlobalmem

INF: Windows Applications Should Not Use EMS Memory

Article ID: Q76689

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

One of Windows's advances over MS-DOS is its ability to have code loaded for more than one application simultaneously. Applications that are loaded cooperate and share the processor and screen.

Under Windows versions 1.x and 2.x, getting Windows, DOS, and the applications to fit into memory simultaneously was a feat of software engineering. Windows made EMS memory [memory made available through the use of expanded memory specification (EMS)] calls to manage memory. If a particular application needed to manage a large memory space, the application was also allowed to make EMS memory calls.

Windows 3.0 exploits the protected modes of the 286 and 386 chips to provide much more memory for applications to use. Windows manages both expanded and extended memory for applications that should no longer need to manage EMS memory for themselves.

Microsoft strongly recommends that Windows applications do not make EMS memory calls to manage expanded memory. Direct application EMS memory management will be removed from future versions of Windows. Non-Windows (DOS) applications that are run in the Windows environment are not affected by this recommendation.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrMmRealmode

INF: Memory Access Methods for Protected Mode Applications
Article ID: Q77226

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Applications that will run solely in one of Windows protected modes (standard mode or enhanced mode) can take advantage of the attributes of protected mode to reduce the size and improve the speed of the application.

These techniques do not apply to applications that can be run in Windows real mode.

More Information:

To assure compatibility with future versions of Windows, an application should not make any assumptions about the protection ring of memory selectors the system provides for memory allocations.

Using global memory can be more straightforward when an application is developed for the protected-mode environment. The following four steps provide a procedure for using memory:

1. Use the GlobalAlloc() function to allocate memory. Applications should ALWAYS use the GMEM_MOVEABLE attribute, which signifies that the linear address of the memory block can be changed by the system. The selector or handle to this block will not change unless the application calls GlobalRealloc() to modify the handle, and changes the number of 64K blocks required to satisfy the request. For example, the selector can change if a 60K block is increased to 70K or if a 70K block is reduced to 60K.
2. Use the GlobalLock() function to obtain the corresponding selector. In protected mode, there is no need to bracket each use of an object with GlobalLock/GlobalUnlock calls. These calls are required in real mode because the Windows memory management algorithm cannot move locked objects in memory. In protected mode, locked objects can be moved without changing the selector value that is used to refer to the object.
3. Use the selector and the range of offsets (from zero to the size of the block) to access the memory.
4. When the memory block is no longer required, unlock the memory block using the GlobalUnlock() function, and free the selector using the GlobalFree() function.

Some applications introduce incompatibilities by implementing a private version of the GlobalLock() function to translate a handle to a selector. While a private function can be made to work for any one

version of Windows, it is not guaranteed to work in future versions.

The four steps above may be used for discardable memory. However, Windows cannot discard the memory while it is locked. Therefore, even in protected mode, applications that use discardable memory objects should unlock each object when it is not in use. This makes these memory blocks candidates for discarding should the system run out of memory.

Selectors should not be shared between applications unless the rules outlined in the dynamic data exchange (DDE) specification are followed. In particular, the `GMEM_DDESHARE` or `GMEM_SHARE` flag must be specified in memory allocation requests. Future versions of Windows may implement separate address spaces; any applications that improperly share memory will not function properly in any such release.

An application that uses any of these techniques should specify the Resource Compiler `-T` switch when resources are added to the application. This will prevent the application from running in real mode.

Additional techniques to decrease the size and increase the speed of an application developed for protected mode can be found by searching on the words:

`prod(winsdk)` and `streamlined`

Additional reference words: 3.00

KBCategory:

KBSubcategory: `KrMmGlobalmem`

INF: Rules for Using Far Pointers to Memory Objects

Article ID: Q77473

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

In Windows version 3.0 standard and enhanced modes, rules for using far pointers have been relaxed substantially. This is due to the use of the protected mode of the 80286, 80386, and 80486 processors. In this mode, far pointers are no longer a segment:offset value. Protected mode far pointers are made up of a selector:offset value. The selector is an index into a descriptor table. Each descriptor contains information about a block of memory, such as size, location in memory (linear memory if in enhanced mode, physical memory if in standard mode), and access rights. For more information on the descriptor table, refer to Chapter 3 of "The 80386/80486 Programming Guide" by Ross P. Nelson (Microsoft Press).

The protected mode of the 80286, 80386, and 80486 processors allows memory blocks to be moved in memory without invalidating their selector:offset value. This is accomplished by changing the reference to the location in memory in the descriptor rather than changing the actual selector value.

Because of this functionality, using far pointers in standard and enhanced modes is much easier. In the following four circumstances, far pointers to data can be assumed to be valid:

1. Global memory that is allocated with GlobalAlloc as movable and nondiscardable

The far pointer returned from GlobalLock can be assumed to be valid as long as the GlobalUnlock function has not been called. Once GlobalUnlock has been called, the far pointer can no longer be assumed to be valid because the memory may be discarded. However, the far pointer will remain valid in the case of memory that is nondiscardable and is not an automatic data segment. Because of the selector technology discussed above, movable memory can move without invalidating the far pointer.

Note: If a block is reallocated using GlobalReAlloc and the new size requires a different number of selectors (for example 50K reallocated to 110K or 65K reallocated to 63K), the base selector value may be changed. Each selector can refer to a maximum of 64K of memory.

2. Far pointers to static or global data within an application

Static and global data within an application is stored in the static data area of the application's DGROUP. Because the whole DGROUP segment is moved, the far pointer will still be valid after

a move in protected mode.

3. Automatic data within a function as long as that function has not been exited

Far pointers to automatic data will be valid as long as no return has been executed from the function that allocated the data. Automatic data is stored on the stack. When an application returns from a function, the memory allocated by that function is no longer allocated and cannot be assumed to be valid. This then makes memory that a far pointer references subject to corruption.

4. Local memory that has been locked with LocalLock

Memory allocated with LocalAlloc comes from the local heap. Memory on the local heap can be moved around within the DGROUP segment, which will cause the offset value of its location to change. This will invalidate any far pointers to the memory. The LocalLock function fixes the memory within the local heap.

For compatibility with future versions of Windows, far pointers to data within an application should NEVER be passed to other applications. The current version of Windows uses one LDT (local descriptor table) for all descriptors associated with data. In the future, one LDT may be used for each application's data, while a GDT (global descriptor table) may be used for shared data. If a selector:offset combination from one application is used in another application in such an environment, the selector will be used to reference a descriptor in the called application's LDT. The descriptor in the called application's LDT will most likely not contain the correct reference for the memory block. The only supported way to share data between applications is global memory allocated with the GMEM_SHARE (also known as the GMEM_DDESHARE) flag.

Selectors that are aliased, using AllocDStoCSAlias or AllocSelector, will not be updated when the memory is moved. For this reason, far pointers that include aliased selectors cannot be assumed to be valid unless the memory has been fixed into place with GlobalFix.

For a more detailed description of memory management under protected mode and Windows 3.0, see Chapters 17 and 18 of "Peter Norton's Windows 3.0 Power Programming Techniques." This book is available from Bantam Computer Books and is coauthored by Paul Yao.

Additional reference words: 3.00 3.0

KBCategory:

KBSubcategory: KrMmGlobalMem

INF: GlobalReAlloc() and GMEM_ZEROINIT Clarified

Article ID: Q92942

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

GlobalReAlloc() is documented in the Windows API's reference manuals. This is a clarification of one of the flags you can set for GlobalReAlloc(). Under one circumstance, when GlobalReAlloc() is used with GMEM_ZEROINIT, it may not zero out all of the reallocated memory. This situation occurs when GlobalReAlloc() is called to shrink a block of memory and then enlarge it.

More information:

When GlobalReAlloc() is used with GMEM_ZEROINIT to increase the size of a block of memory in the global heap, it will zero out only the bytes it adds to the memory object; it does not initialize any of the memory that existed before the call.

Windows allocates memory from the global heap in multiples of 32 bytes; enhanced mode allocates memory on even 32-byte boundaries, and standard mode allocates memory on odd 32-byte boundaries (that is, /E allocates 32/64/96 bytes, /S allocates 16/48/80 bytes). Thus, when 10 bytes are requested, enhanced Windows actually allocates 32 bytes; when 55 bytes are requested, enhanced Windows allocates 64 bytes.

Suppose we have the following sequence of calls in Windows enhanced mode:

```
HGLOBAL hMem ;

    // 32 bytes are actually allocated, not 10 because Windows
    // allocates global memory in multiples of 32 bytes. All
    // 32 bytes are initialized to zero.
hMem = GlobalAlloc(GMEM_ZEROINIT | GMEM_FIXED, 10);

    // Here, we allocate 32 more bytes and add them to the end of
    // the first 32 bytes. ReAllocating to 40 bytes will cause the
    // block to be 64 bytes long. Only the second 32
    // bytes are initialized to zero. The first 32 bytes are left
    // alone.
hMem = GlobalReAlloc(hMem, 40, GMEM_ZEROINIT | GMEM_FIXED);

    // Copy 39 bytes into the memory block. The first 39 bytes
    // will contain the string.
lstrcpy((LPSTR)GlobalLock(hMem), "This is a big enough string for our job")

    // Now we shrink the block to 10 bytes. After the call, the
    // block will be 32 bytes long; the second 32 bytes are freed
    // and will no longer exist. The first 32 bytes will still
```

```
        // contain the same characters as before the call.
hMem = GlobalReAlloc(hMem, 10, GMEM_ZEROINIT | GMEM_FIXED) ;

        // Now, we enlarge the block back to 40 bytes. After the call,
        // the block will be 64 bytes long, and the second 32 bytes
        // will be initialized to zero. The first 32 bytes will be left
        // alone, however. The area between bytes 10 and 32 does *not*
        // get initialized!
hMem = GlobalReAlloc(hMem, 40, GMEM_ZEROINIT | GMEM_FIXED) ;
```

When GlobalReAlloc() is called to shrink the block, we told it that we wanted only 10 bytes; that's all we should use. Then when we enlarge it back to 40 bytes, GlobalReAlloc() only initializes the memory it adds to the current block--which is from bytes 33 to 64. The bytes between 10 and 40 were previously used, but GlobalReAlloc() did not initialize them because it did not allocate them.

As a result, applications that call GlobalReAlloc() to shrink and then re-enlarge a block of previously used data should not expect that all the bytes will be initialized to zero.

Additional reference words: 3.0 3.00 3.10 3.1
KBCategory:
KBSubcategory: KrMmMemattribs

INF: Large Model and Windows 3.0 Protected Mode
Article ID: Q79275

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

Large model applications can have more than one data segment. Windows fixes the extra data segments of applications that have more than one data segment even if they are declared MOVEABLE in the module definition (DEF) file. Normally, these fixed data segments are movable by Windows's memory manager in protected mode. However, there is a problem, which causes fixed segments to be page locked, with the Windows 3.0 memory manager. When a segment is page locked, the physical memory associated with the segment becomes an obstacle for the memory manager because the segment cannot be moved in physical memory, nor can it be paged to disk. This behavior can have a serious impact on system performance.

There are other issues to consider with respect to large model applications. An application compiled with large model runs much slower than an application compiled with medium or small model due to FAR variables accesses. Large model applications use far pointer arithmetic, which is generated by the compiler when the -AL compiler switch is used, or when a far pointer is explicitly declared in the application code.

Also, Windows cannot run more than one instance of an application that has more than one data segment, because Windows's kernel cannot presently resolve fix-ups to multiple data segments. For these reasons, it is strongly recommended that Windows applications use only small or medium memory model, which uses only one data segment.

More Information:

A large model application is limited to one instance because the Windows kernel presently cannot resolve fix-ups to multiple data segments. Consider the following code fragment that establishes DS:

```
    mov  ax, _data_01
    mov  ds, ax
```

This code is shared by all instances of the application. When the code is loaded, only one value can be put into DS. Windows has no way to "glue," or associate, other data segments to a given instance of the application.

Windows's determination to allow only one instance is made when the loader scans the EXE header of the application. Upon discovery of more than one data segment, the application is limited to one instance.

In summary, there are three problems with applications compiled with large model:

1. Large model applications use far pointer arithmetic, and thus will run

more slowly.

2. Large model applications that have more than one data segment will have their extra data segments page locked by Windows and cause the degradation of Windows's memory management.
3. Large model applications that have more than one data segment can run only one instance.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrModlsLargemod

INF: C Run-Time Functions Can Use Far Pointers in Medium Model
Article ID: Q66462

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Far pointers to data can be used in calls to the C run-time library routines written using the medium model. This is often necessary because many Windows applications are written using the medium model, but must pass far data pointers to the medium model C run-time library routines. Unless precautions are taken, passing far pointers to medium model C run-time routines will fail.

To use far pointers in calls to medium model C run-time routines, the model-independent version of the C run-time functions must be explicitly specified. A model-independent version of a C run-time function requires specification of the size of the data pointers required (NEAR or FAR) for both function parameters and return values.

Not all C run-time routines have model-independent versions. To determine if a routine has a model-independent version, please consult the header file associated with the routine or the C run-time source manual.

More Information:

When an application is compiled using the medium memory model, the C compiler assumes that the application will have one data segment and multiple code segments. Because the application has only one data segment, all pointers to data are assumed to be near pointers. Therefore, when the compiler encounters a C run-time function in the source code, it automatically assumes that any pointer parameters contain near pointers. The compiler uses the appropriate medium model declaration for the run-time functions. This is appropriate for NEAR data items, but many Windows API functions require or return FAR pointers, such as GlobalLock().

To override the compiler's assumptions specify the model-independent version of the desired routine in the application source code. The header file associated with the routine or the C run-time source manual can be used to determine which C run-time routines have model-independent versions. Typically, the model-independent versions of C run-time functions are preceded by an "_n" or an "_f". For example, strdup(), a memory model-dependent function, has two model-independent derivatives: _nstrdup() for NEAR pointers and _fstrdup() for FAR pointers.

When using the model-independent versions of the C run-time routines, the compiler will not assume that the application's data is near. Thus, far pointers can be used in medium model applications where near pointers would normally be used.

Unfortunately, not all C run-time routines offer this flexibility. If the routine does not have a model-independent version, two options are available:

1. Write a model-independent routine that offers the same functionality as the C run-time routine.
2. Copy the data into the default data segment so that near pointers, and the standard C run-time routines, can be used.

One C run-time routine that demonstrates model independence is `strncpy()`. If the application source code includes a line similar to the following

```
Char_ptr = strncpy(String1, Const_String2, Count);
```

the compiler will use the default declaration for the routine. That declaration is found in the header file and resembles the following:

```
char *strncpy(char *string1, const char *string2, size_t);
```

In this situation, the data must be in the default data segment because the routine will use the DS register when referencing both strings.

In the same medium model applications, if one or both of the strings are in a data segment other than the default data segment, modify the same source line as follows:

```
Char_ptr = _fstrncpy(String1, Const_String2, Count);
```

In this case, the compiler will use the following function declaration

```
char _far * _far _fstrncpy(char _far *string1,  
                           const char _far*strings,  
                           size_t count);
```

and the application will then properly access the data in the far segments.

Note: When using the same C run-time routine in a large model application, the default function declaration will be the model-independent version. Thus, in the example above, the compiler will replace the call to `strncpy()` with an appropriate call to `_fstrncpy()`.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrModlsCusted

INF: Windows 3.0 Does Not Support Static Data Segments > 64K
Article ID: Q67707

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

Windows version 3.0 does not support static data segments that are larger than one memory segment (64K). This limitation has been removed from Windows version 3.1. In Windows version 3.0, Windows reserves the right to load any segment from the executable file into any location in memory. Because the loader does not necessarily load consecutive segments into contiguous memory, data structures such as huge arrays that require consecutive memory blocks are not guaranteed to be loaded correctly.

Under Windows 3.0, if an application requires a huge array, allocate the array dynamically, at run time, rather than statically, at compile time. This can be done by using the GlobalAlloc function to obtain a memory block that is accessed through a huge pointer.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrModlsCusted

INF: Sample Code Unlocks Large-Model Extra Data Segments

Article ID: Q83363

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

An application that is compiled using the large memory model can have multiple code segments and multiple data segments.

Under Windows 3.0, all the data segments except for the first one (the so called "extra" data segments) are loaded into memory segments that are fixed and page locked. Fixed and page locked segments interfere with effective memory management. They reduce the amount of free memory addressed below 1 megabyte (MB) in the linear address space, and in some cases, cause out-of-memory errors as additional applications are run.

Under Windows 3.1, extra data segments are loaded into movable memory and the above difficulties with fixed and page locked segments do not apply.

A large model application can change the attributes of its extra data segments by calling the GlobalPageUnlock and GlobalUnfix functions. However, to do this effectively, the application must be able to enumerate the selectors for its extra data segments.

LARGEAPP is a file in the Software/Data Library that demonstrates using the GlobalPageUnlock and GlobalUnfix functions to change the attributes of an application's extra data segments. LARGEAPP contains the source code to two applications: LARGEAPP, which is a large-model application for the Windows environment, and SYMSEG, which is an MS-DOS (non-Windows) application.

SYMSEG is a utility that reads a symbol file produced by the Microsoft Linker, and creates a table of the data segments in an application. LARGEAPP uses the information from this table to enumerate its own data segments.

LARGEAPP determines the version of Windows under which it is running. The segment attributes must be changed only if the application is running under Windows 3.0.

LARGEAPP can be found in the Software/Data Library by searching on the word LARGEAPP, the Q number of this article, or S13378. LARGEAPP was archived using the PKware file-compression utility. The Microsoft C Compiler version 6.0 and the Microsoft Macro Assembler (MASM) version 6.0 are required to build the files in LARGEAPP and to use the techniques of this sample.

Additional reference words: 3.00 3.10 softlib LARGEAPP.ZIP
KBCategory:
KBSubcategory: KrModlsLargemod

INF: Using Large Memory Model, Microsoft C/C++, & Windows 3.1
Article ID: Q90294

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

In Windows version 3.0, using the large memory model with Microsoft C version 6.0 is not recommended. For more information on the problems that occur when using the large memory model under Windows 3.0, query this knowledge base on the following words:

large and model and protected

Windows version 3.1 and the Microsoft C/C++ version 7.0 compiler have resolved most of these problems and make the large memory model much more suitable for developing applications under Windows 3.1.

More Information:

There are three main problems when using the large memory model under Windows:

1. Large model applications can have multiple data segments (this is always the case when compiling with Microsoft C 6.0). In Windows 3.0, these extra segments are loaded as FIXED and page-locked. Having a large amount of FIXED page-locked memory results in a serious degradation of the memory manager's performance. FIXED memory is allocated from the bottom of the global heap, which means it normally lies in conventional memory (below 1 MB), which is needed by the Windows loader when it loads a new module into memory. For more information on this topic, query this knowledge base on the following words:

using and memory and below and megabyte

This problem has been corrected in Windows 3.1. In Windows 3.1, an application's code and data segments are always loaded as MOVEABLE, regardless of the segment attributes specified in the application's definition (.DEF) file. A DLL's code and data segments are loaded exactly as they are specified in the DLL's .DEF file.

2. Under both Windows 3.0 and Windows 3.1, only one instance of an application with multiple data segments can be run at one time. This is because the Windows loader can't fix-up multiple instances of a far pointer because code is shared among all instances of an application.

In most cases, this problem may be resolved by using the /Gx and /Gt compile options with the Microsoft C/C++ 7.0 compiler. The /Gx option causes the compiler to force static data into the default

data segment, which results in the application having only one data segment. Note, however, that this works only if the application's static data, string literals, stack, and heap all fit into a 64K segment.

The /Gt switch specifies the size a static object must be to be allocated its own data segment. When trying to create a single data segment, this threshold value should be large enough to ensure that no data objects are allocated outside of the default data segment. If the /Gt option is not specified, the data threshold will be 32767.

3. The final problem with using large model under both Windows 3.0 and Windows 3.1 is the performance degradation that occurs when an application uses far pointers to access its data.

When running under the protected mode of the Intel 286, 386, and 486 processors, a far address is a combination of a selector and an offset. A selector is essentially an index into an array called a descriptor table. A descriptor is an 8-byte value that contains information about the segment, such as its base address, size, read/write privileges, and so forth.

Whenever a far variable is referenced in protected mode, the processor must load the descriptor into one of the segment registers' descriptor cache and mark the descriptor as being accessed. This means that a reference to a far variable requires at least two reads from and one write to memory just to obtain the variable's linear address.

Adding this delay to the time needed for performing far pointer arithmetic amounts to a significant loss of performance when compared to the medium model, which uses near pointers for data. One method to increase the performance of a large model application is to explicitly declare commonly used global or static variables as being NEAR, which causes the compiler to allocate the variable in the default data segment, and to use a near pointer when referencing the variable.

Conclusion:

Because Windows 3.1 and Microsoft C/C++ 7.0 have resolved most of the problems related to use of the large memory model, developing application for Windows 3.1 with the large memory model is much more acceptable than it was under Windows 3.0 and Microsoft C 6.0.

Additional reference words: 3.00 3.0 3.10 3.1 6.00 6.0 7.00 7.0

KBCategory:

KBSubcategory: KrModlsLargemod

FIX: Using fputc() or fgetc() in Large Model DLL UAEs
Article ID: Q77476

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9110012

SYMPTOMS

When the function fputc or fgetc is used in a Windows dynamic-link library (DLL) created with the LDLLCEW.DLL library, the application experiences an unrecoverable application error (UAE).

CAUSE

The functions are retrieving a near pointer from the stack when a far pointer is required.

STATUS

Microsoft has confirmed this to be a problem in the Microsoft C large-memory-model DLL run-time library provided with the Windows Software Development Kit version 3.0. This problem was corrected in the Microsoft C/C++ Optimizing Compiler version 7.0.

Additional reference words: 3.00 7.00

KBCategory:

KBSubcategory: KrModlsNearfarcls

INF: Differences Between Task Handles and Instance Handles

Article ID: Q76676

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Windows creates two handles associated with each task running in the system. One handle is the instance handle, `hInstance`, which is passed to the `WinMain` function when the program starts execution. The other handle is the task handle, `hTask`, which is returned by `GetCurrentTask`.

The numerical values of `hInstance` and `hTask` are different. In some routines, `hTask` can be used in place of `hInstance` without any problems. In other routines, using `hTask` in place of `hInstance` causes incorrect behavior and may result in unrecoverable application errors (UAEs). Using `hTask` in place of `hInstance` is considered bad form and will probably cause problems when the application is run under future versions of Windows.

To retrieve the instance handle for the application currently running, use the following code fragment:

```
hInstance = GetWindowWord(GetActiveWindow(), GWW_HINSTANCE);
```

To retrieve the task handle for the application currently running, use the following code fragment:

```
hTask = GetCurrentTask();
```

More Information:

The instance handle, `hInstance`, is used to identify the data associated with a particular instance of an application. The task handle, `hTask`, is the handle to a structure, called the task database (TDB), which contains information about the task (for example, its queue, module handle, and so forth). The instance handle and the PDB (program database), also known as the PSP (program segment prefix), are also stored in the task database. The `GetCurrentPDB()` function returns a handle to the current PDB.

Each instance of an application has both an instance handle and a task handle. Dynamic-link libraries (DLLs) are not tasks; therefore, they have only an instance handle and do not have a task handle.

Additional reference words: 3.0 3.00

KBCategory:

KBSubcategory: KrTskinsDtabse

INF: HANDLEs Returned by GetModuleHandle and LoadLibrary
Article ID: Q78327

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

The `GetModuleHandle` and `LoadLibrary` functions each return a `HANDLE` datatype as documented; however, these two functions have distinct purposes.

`GetModuleHandle` returns a module handle [the handle to the PSP (program segment prefix)].

`LoadLibrary` calls the `LoadModule` function with a null `lpParameterBlock` parameter. If the name of a library is specified in a call to `LoadLibrary`, a module handle is returned. If the name of an executable file is specified in the call to `LoadLibrary`, an instance handle is returned.

In either case, functions that require a `HANDLE`-type parameter, such as `GetProcAddress`, will recognize both types of handles.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrTskSinsDtabse

INF: Retrieving the Names of Simultaneous Tasks Under Windows
Article ID: Q80124

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

There are situations when it is necessary for an application to obtain a list of all applications that are running in the Windows environment at a particular time. Instead of using the EnumWindows() function and an application-supplied callback function to enumerate all parent windows, the application can retrieve a handle to the first window in the task list and walk through the list to obtain the names of all windows in the task list.

More Information:

The most efficient way to retrieve the name of each task running under Windows is to use the GetWindow() function. GetWindow(hwnd, GW_HWNDFIRST) provides the handle to the first window in the task list. The application can walk through the task list by calling GetWindow(hwndCurrent, GW_HWNDNEXT). The following example demonstrates how to obtain a handle to each top-level window. The GetWindowText() function provides the name of each window from its handle.

```
hwndNext = GetWindow(hWnd, GW_HWNDFIRST);
while (hwndNext)
{
    if ((hwndNext != hWnd) && // Do not get this application's
        // name.
        (IsWindowVisible(hwndNext)) &&
        (!GetWindow(hwndNext, GW_OWNER)))
    {
        if (GetWindowText(hwndNext, (LPSTR)szTemp, sizeof(szTemp)))
        {
            // This is a valid top-level window handle.
            // Its name is in szTemp...
        }
    }
    hwndNext = GetWindow(hwndNext, GW_HWNDNEXT);
}
```

The code above will retrieve the name of each visible window. To also retrieve the names of invisible windows, remove the call to IsWindowVisible().

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrTskinsModnam

INF: Heap and Stack Usage Within Windows

Article ID: Q10641

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

The following information clarifies heap and stack usage within a segment versus heap and stack usage within the application as a whole, for Windows versions 2.x and 3.0.

An application gets a single default data segment, from which the stack is allocated. The remainder of the data segment is used for static data and the dynamic local heap.

The STACKSIZE keyword in the .DEF file specifies the size of the application's stack; it is allocated from DGROUP and therefore is limited to a maximum size of (64K - heap size - static data) bytes. The stack size is not dynamically enlarged or reduced.

The HEAPSIZE keyword in the .DEF file specifies the initial default local heap size. Windows attempts to enlarge the heap size when local allocations fail; however, the heap is limited to a maximum size of (64K - stack size - static data).

The sum of the static data, stack, and local heap cannot exceed 64K.

Multiple local heaps can be managed using the LocalInit() call and swapping the DS register as needed. For more information on this technique, query on the following words:

handle and limit and globalalloc and register
Chapter 18 (pages 707-724) of "Windows 3.0 Power Programming Techniques," by Paul Yao and Peter Norton (Bantam Computer Books), contains more details about this technique.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrTskinsMisc

INF: Why WinExec() Returns Error Code 8: Insufficient Memory
Article ID: Q66391

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

When the value 8 is returned from the WinExec() function, it indicates "insufficient memory to start application." This is the same condition indicated by the value 8 returned from MS-DOS Interrupt 21h Function 4Bh. This return value is not listed on page 4-459 of the "Windows Software Development Kit Reference Volume 1," version 3.0.

The WinExec() function is used to start a Windows or non-Windows application from inside a Windows application. If the value returned from WinExec is greater than 32, the application has been started successfully; otherwise, the value returned is an error code.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrTskinsSpawn

INF: Windows: Nonpreemptive vs. Preemptive Scheduling

Article ID: Q11248

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

Preemptive scheduling, which Windows DOES NOT do, is defined in the following way:

Between any two application instructions, N instructions may execute in another application's context, where N is greater than or equal to zero.

A nonpreemptive system, such as Windows, will guarantee that this number N will always be zero.

In nonpreemptive scheduling, an application is not forced out of context asynchronously (that is, it is not preempted). Instead, the application runs until it explicitly gives up control. Windows-aware applications give up control through various system calls. Although they are not aware of it, DOS applications give up control whenever they attempt various I/O functions.

DOS applications running under Windows 3.0 are in fact preemptively multitasked. In contrast, all WINDOWS applications are nonpreemptively multitasked. When the system is viewed from a Win386 perspective, Windows runs in the system virtual machine (VM) and that VM competes for time slices along with the rest of the DOS applications running in other virtual machines. Keep in mind that unlike DOS applications, all windows applications run inside the system VM, and are not given their own virtual machine.

Note: An interrupt is not considered to be a form of preemption unless there is an application context switch during the interrupt. An interrupt takes the execution stream into the kernel, which returns back to the same place without running another application, much in the same way a call would.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrTskinsMisc

INF: How to Determine When Another Application Has Finished
Article ID: Q67673

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

Two separate applications sometimes need to cooperate in the Windows environment. Two Windows applications may work in tandem, or a Windows application may require the services of a non-Windows application.

This article examines the issues involved when a Windows application requires notification that another application has completed its processing.

More Information:

First, the following constraints should be considered:

1. Windows was not designed to synchronize the operation of a Windows application with a non-Windows (MS-DOS) application in any mode (real, standard, or 386 enhanced).
2. Windows does not provide any automatic way of determining if another application has finished, or was run correctly. This first section included below discusses techniques that can be included in code that is written for cooperating applications (for example, when both are Windows applications, and when one is for Windows and the other is not). The second section listed below discusses techniques to apply when the other application is beyond the programmer's direct influence.

Techniques to Use for Cooperating Applications

The following are two options that can be used if the programmer is developing both applications and each runs under Windows:

1. The two applications can communicate through a dynamic data exchange (DDE) messaging protocol that the programmer establishes. DDE is explained in Chapter 22 of the "Microsoft Windows Software Development Kit Guide to Programming," version 3.0.
2. Each application can register the same application-specific message text with Windows via the RegisterWindowMessage() function and receive a numeric value for the message. The terminating application then sends this value to the other application. This can be accomplished by broadcasting the message to all windows in the system. See the documentation for the PostMessage() function in the "Microsoft Windows Software Development Kit Reference Volume 1" for more information.

When one of the cooperating applications that is being developed does not run under Windows, the following must occur:

1. When the MS-DOS application completes or encounters a fatal error, it should write a message into a specified file in the TEMP directory.
2. The Windows application should then perform the following steps:
 - a. Use the SetTimer() function to create a system timer that will fire at desired intervals. The estimated completion time of the function is one possible interval. Another would be "estimated time to first possible failure" if the MS-DOS application will be writing an error string for the Windows application to display.
 - b. Initiate the execution of the MS-DOS application with the WinExec() function and continue with any normal processing.
 - c. Upon receipt of the timer message, the TEMP directory can be checked to determine if a message file is present. If the file is present, the message in the file is parsed to see if termination was due to success or failure.

Techniques to Use for Third-Party Applications

For a third-party application, the steps taken must access information that is entirely external to the MS-DOS application. One possible way to implement this method is to use the title of the WINOLDAP window to determine if the application is still running. In this case, the following steps should be taken:

1. Use the SetTimer() function to create a system timer. Timer messages should be at least a few seconds apart. This allows WINOLDAP to create its window and begin processing.
2. In real and standard modes, Windows application processing stops while the MS-DOS application is processing. In enhanced mode, Windows' behavior depends on the settings in the program information file (PIF) that corresponds with the MS-DOS application. For more information about the allocation of the processor when an MS-DOS application is running, query on the following keywords:

prod(winsdk) and WinExec() and dependent
3. When the timer message is received, the FindWindow() function is then used to search for the caption of the MS-DOS application's window. The caption is created from the "Window Title" section of its PIF file, or if it is blank or not found, the filename of the old application. If the caption is no longer present, the application is deemed to have completed its processing.

Another possible solution is to create a batch file in MS-DOS that checks for error level information returned from the program, and then creates files in the TEMP directory. The Windows application can then

check for these "result" files to determine the MS-DOS application's status.

The following batch file creates a sentinel file named BEGIN.TMP. Until this file is deleted, the MS-DOS application is considered to be running. Successful completion creates the result file, END.TMP, and then BEGIN.TMP is deleted. An execution error creates the result file named STOP.TMP, and then removes BEGIN.TMP.

```
1: echo start > %TEMP%\begin.tmp
2: MyDosApp
3: if errorlevel 1 goto bad
4: if errorlevel 0 goto good
5: goto end
6: :bad
7: echo bad > %TEMP%\stop.tmp
9: goto end
10: :good
11: echo good > %TEMP%\end.tmp
12: goto end
13: :end
14: del begin.tmp
15: goto end
```

A system timer is employed as above to direct the Windows application to check for the existence of the sentinel and result files.

If it is necessary for a MS-DOS application to communicate with Windows, then contact the SoftBridge company. SoftBridge can provide additional information on its product offerings. Contact information is listed on page 107 of the "Windows Shopping" book, supplied with the Windows version 3.0 retail product.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrTskSinsMonitor

INF: The Purpose of WINSTUB in Windows SDK

Article ID: Q11591

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

WINSTUB is provided with the Windows Software Development Kit (SDK) as a normal DOS program. It contains only an assembly equivalent to a printf() line. Since it is assembly, it does not have the overhead it would if it used the C runtime library. WINSTUB can be used as a stub if no DOS version of a program exists. To have both a Windows version and a DOS version of a program in one EXE file, replace the following line in the DEF file

```
STUB 'WINSTUB.EXE'
```

with the following line

```
STUB 'MYDOSAPP.EXE'
```

and relink.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrTskmsinsMisc

INF: Sample Code Spawns Task and Waits for its Termination
Article ID: Q84456

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.0 and 3.1
-

Summary:

TERMWAIT is a file in the Software/Data Library that demonstrates how an application can launch a child task and then wait for it to complete before executing specific code. TERMWAIT uses notification messages from the Toolhelp dynamic-link library (DLL) to determine that the child task has completed. The techniques demonstrated by the TERMWAIT sample work for both Windows and MS-DOS (non-Windows) child tasks.

TERMWAIT can be found in the Software/Data Library by searching on the keyword TERMWAIT, the Q number of this article, or S13429. TERMWAIT was archived using the PKware file-compression utility.

More Information:

The TERMWAIT sample application calls the NotifyRegister function to install a notification callback function before it calls the WinExec function to launch the child task. If a callback function is registered, it is called before any task in the system terminates. The notification callback function calls the TaskFindHandle function to fill a TASKENTRY data structure with information about the terminating task. When the callback determines that the child task has terminated, it notifies the TERMWAIT application.

When it spawns the child task, TERMWAIT sets its bChildIsExecuting global variable to TRUE. The notification callback resets this variable to FALSE when the child task is complete. Any code that must not execute while the child task is running can query the value of the bChildIsExecuting flag. During the wait, any menu selections that will cause reentrancy problems should be disabled. Doing so keeps the user informed about the options that are presently valid. In the TERMWAIT sample, the AfterChildHasTerminated function contains code that is executed only after the child task has completed.

If an application tracks a number of child tasks, its NotifyRegister callback function should process both the NFY_STARTTASK and NFY_EXITTASK notifications. The callback function uses these notifications to maintain a list of child task handles. Note that while no two tasks will have the same handle, it is possible for task handles to be reused. Consequently, if one task ends and a new task begins, the new task can use the same task handle.

Version 3.1 of the Windows Software Development Kit is required to build the TERMWAIT sample. However, the compiled code is compatible

with both Windows 3.0 and 3.1. Note that because the TOOLHELP.DLL is not part of the default installation for Windows 3.0, it must be installed into the Windows system directory (by default, C:\WINDOWS\SYSTEM) before TERMWAIT will run. The TOOLHELP.DLL file is provided as a redistributable file with the Windows 3.1 SDK.

Additional reference words: 3.00 3.10 softlib TERMWAIT.ZIP synchronize

KBCategory:

KBSubcategory: KrTskSinsSpawn

BUG: GetModuleFileName Returns Relative File Path

Article ID: Q85330

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
-

SYMPTOMS

=====

When the GetModuleFileName() function returns a reference to a dynamic-link library (DLL) file, the reference is relative (not fully qualified) under the following circumstances:

- One of the directory references in the MS-DOS PATH environment variable is relative. Assume that the relative directory reference refers to drive X.
- An application is implicitly linked to a DLL. The DLL is installed in the current (default) directory of drive X.
- The application is installed in a directory other than the one in which the DLL is installed.

STATUS

=====

Microsoft has confirmed this to be a problem in Windows version 3.1. We are researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

MORE INFORMATION

=====

The following describes the situations under which this problem occurs:

- The PATH statement in the AUTOEXEC.BAT file resembles the following:

```
PATH=C:\DOS;C:\WINDOWS;D:.;C:\APPS
```

The relative reference in this path refers to drive D.

- Run an application from the C:\WINDOWS directory that implicitly links to a DLL that is stored in the current (default) directory of the D drive.
- If any application calls the GetModuleFileName() to retrieve the file name for the DLL, the function returns a relative reference to the file (for example, D:.\DLL.DLL).

Additional reference words: 3.10
KBCategory:
KBSubcategory: KrTskinsModnam

PRB: Avoiding
Article ID: Q86230

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

SYMPTOMS

During the process of loading an application, Microsoft Windows displays an application execution error message box with the following message:

Insufficient memory to run this application; close one or more Windows applications and try again.

CAUSE

One of the reasons this message is displayed is that the system has no memory available with an address less than 1 megabyte (MB). When Windows loads an application, it calls the GlobalDosAlloc function to allocate memory in the address space below 1 MB for the application's task database. If the GlobalDosAlloc call fails, Windows displays the application execution error message.

RESOLUTION

Use as little memory below 1 MB as possible.

More Information:

A common situation that leads to an insufficient memory error regards an application developed using a large memory model that includes many extra data segments and/or very large extra data segments. When Windows loads the application, it allocates fixed memory to hold the extra data segments. In Windows 3.0, these fixed memory blocks are also page locked, which prevents the memory manager from moving the blocks to disk as memory fills.

Fixed memory is allocated from the bottom of the global heap, which starts at the lowest possible memory address. If all the memory below 1 MB is filled with page locked memory blocks, Windows cannot move blocks in memory or swap blocks to disk to free any memory. When Windows cannot allocate a task database for a new task, it displays the error message box discussed above.

For more information on the page locking of fixed memory segments in Windows 3.0, query on the following words in the Microsoft Knowledge Base:

prod(winsdk) and buglist3.00 and page and locks

An application can determine whether all memory addressed below 1 MB is in use by attempting to allocate a small block of memory with the GlobalDosAlloc function. (Windows 3.0 requires 512 bytes of low memory

for the task database of each application.) If the allocation fails, Windows will not be able to start another program. Even if the allocation is successful, the failure of another memory allocation required by the application may prevent it from loading.

One way to address this problem is to remove the page lock from an application's extra data segments. However, memory accessed by an interrupt service routine must be page locked to keep its data available at all times. An application can use the services of the tool helper dynamic-link library (TOOLHELP.DLL) and a few Windows functions to modify the flags on the extra data segments. After modification, the data segments are movable and not page locked.

Use the GlobalFirst and GlobalNext functions provided by the tool helper library to walk the global heap looking for memory blocks owned by the application. If a memory block is page locked (the wcPageLock field of the GLOBAENTRY data structure is not zero), call the GlobalPageUnlock function to change the lock count for the memory block. Calling GlobalRealloc to change the block from fixed to movable memory might also be desirable.

For more information on the tool helper library, see Chapter 8 of the "Microsoft Windows Software Development Kit: Programmer's Reference, Volume 1: Overview" version 3.1 manual.

Another method to unlock data segments uses the GlobalPageUnlock and GlobalUnfix functions. For more information on this technique, query on the following words in the Microsoft Knowledge Base:

prod(winsdk) and largeapp
Additional reference words: 3.00 3.10 pagelocked page-locked SDK
KBCategory:
KBSubcategory: KrTskinsSpawn

INF: SpawnAndWait DLL Provides Asynchronous Spawn Function
Article ID: Q105116

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows version 3.1
-

SUMMARY

=====

The sample RUNWAIT demonstrates using TOOLHELP.DLL to provided a dynamic-link library (DLL) function that spawns applications and waits for their termination before returning from the function call. The sample is compatible with Visual Basic (VB) and Windows 3.0. The sample loads a hidden task at DLL startup time to own the Toolhelp callback.

RUNWAIT can be found in the Software/Data Library by searching on the word RUNWAIT, the Q number of this article, or S14345. RUNWAIT was archived using the PKware file-compression utility.

MORE INFORMATION

=====

The following is the VB function declaration (this must be on a single line in VB):

```
Declare Function SpawnAndWait Lib "RUNLIB.DLL" (ByVal parenthwnd%,  
ByVal lpszOp$, ByVal lpszFile$, ByVal lpszParams$, ByVal lpszDir$,  
ByVal nShow%)
```

```
DWORD SpawnAndWait(hwnd, lpszOp, lpszFile, lpszParams, lpszDir, fsShowCmd)
```

```
HWND    hwnd           /* Handle of parent window                */  
LPCSTR  lpszOp         /* Address of string for operation to perform */  
LPCSTR  lpszFile       /* Address of string for filename          */  
LPCSTR  lpszParams     /* Address of string for executable-file parameters */  
LPCSTR  lpszDir        /* Address of string for default directory  */  
int     fsCmdShow      /* Whether file is shown when opened       */
```

The SpawnAndWait function executes and waits for termination of the specified application or associated file.

Parameter Description

hwnd	Identifies the parent window. This window receives any message boxes an application produces (for example, for error reporting).
lpszOp	Points to a null-terminated string specifying the operation to perform. This string can be "open" or "print". If this parameter is NULL, "open" is the default value.

lpszFile Points to a null-terminated string specifying the file to open.

lpszParams Points to a null-terminated string specifying parameters NULL; "open" is the default value. passed to the application when the lpszFile parameter specifies an executable file. If lpszFile points to a string specifying a document file, this parameter is NULL.

lpszDir Points to a null-terminated string specifying the default directory.

fsShowCmd Specifies whether the application window is to be shown when the application is opened. See ShowWindow for valid values.

Returns

HIWORD == hInstance of started application. Values less than 32 are errors returned from ShellExecute. 0xFFFF is a general error.

LOWORD == Return code of spawned application. 0xFFFF is a general error.

Comments

The file specified by the lpszFile parameter can be a document file or an executable file. If it is a document file, this function opens or prints it, depending on the value of the lpszOp parameter. If it is an executable file, this function opens it, even if the string "print" is pointed to by lpszOp.

WARNING: This function will not wait on applications such as Word and Excel that respond to the DDE broadcast made by ShellExecute or the second instance of multiple data applications.

WARNING: This function supports only one block at a time per task. Calling tasks should not call this function when it has a prior pending SpawnAndWait call.

Sample Code

```
DWORD SpawnAndWaitIndirect(lpSpawnWait)

LPSPAWNWAIT lpSpawnWait /* Far reference to SPAWNWAIT structure

typedef struct tagSPAWNWAIT
{
    HWND    hwnd;
    LPCSTR  lpszOp;
    LPCSTR  lpszFile;
    LPCSTR  lpszParams;
    LPCSTR  lpszDir;
    int     fsShowCmd;
    LPMMSGPROC lpmsgproc;
```

```
} SPAWNWAIT;
```

Member	Description
--------	-------------

hwnd	Handle of parent window
lpszOp	Address of string for operation to perform
lpszFile	Address of string for filename
lpszParams	Address of string for executable-file parameters
lpszDir	Address of string for default directory
fsShowCmd	Whether file is shown when opened
lpmsgproc	Address of application provided MessagePump (must load DS on entry)

```
void CALLBACK MessagePump(lpmsg)
```

```
LPSMG lpsg /* Long pointer to MSG to process
```

Message Proc is a place holder for an application-provided callback function (which must load DS on entry) that will process messages retrieved in RunLib's PeekMessage loop. It allows the calling application to do modeless dialog box and accelerator message processing.

lpmsgproc should be set to NULL if it is not used. RunLib will do a default TranslateMessage/DispatchMessage instead.

The following is an example of a message processing function for a MDI application:

```
void CALLBACK MyMessagePump(LPMSG lpmsg)
{
    if(!TranslateMDISysAccel(hClient, lpsmg) &&
        !TranslateAccelerator(hFram, hAccel, lpsmg))
    {
        TranslateMessage(lpmsg);
        DispatchMessage(lpmsg);
    }
}
```

Returns

HIWORD == hInstance of started application. Values less than 32 are errors returned from ShellExecute. 0xFFFF is a general error.

LOWORD == Return code of spawned application. 0xFFFF is a general error.

Comments

The file specified by the lpszFile parameter can be a document file or an executable file. If it is a document file, this function opens or prints it, depending on the value of the lpszOp parameter. If it is an executable file, this function opens it, even if the string "print" is pointed to by lpszOp.

WARNING: This function will not wait on applications such as Word and Excel that respond to the DDE broadcast made by ShellExecute or the second instance of multiple data applications.

WARNING: This function supports only one block at a time per task. Calling tasks should not call this function when it has a prior pending SpawnAndWait call.

Additional reference words: 3.00 3.10 toolhelp spawn p_wait
synchronous winexec vb

KBCategory:

KBSubcategory: KrTskinsSpawn

FIX: Application with No Exports Crashes Under Windows 3.0
Article ID: Q88857

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.0 and 3.1
-

Summary:

PROBLEM ID: WIN9209001

SYMPTOMS

An application built for Microsoft Windows version 3.0a that has no exported functions runs correctly under Windows version 3.1, but crashes upon loading under Windows version 3.0a.

CAUSE

Due to a problem in Windows versions 3.0 and 3.0a, if an application has no exports, it may crash upon loading depending on the size of the path name, the size of the resources, and the size of the executable file. Typical symptoms include a general protection (GP) fault immediately after the program is started, even before your application's WinMain function is called.

This problem typically shows up only in small "stub" programs (for instance, programs that are used to load other applications).

RESOLUTION

Add at least one exported function to the EXPORTS section of your .DEF file. The function name you add to the .DEF file must also have an identically named function in your source files. If necessary, this exported function can be a dummy function that simply returns.

STATUS

Microsoft has confirmed this to be a problem in Windows versions 3.0 and 3.0a. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00 3.00a 3.10

KBCategory:

KBSubcategory: KrTskmsinsMisc

INF: Callback Functions in Multiple Instance Applications

Article ID: Q102871

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, versions 3.0 and 3.1
-

SUMMARY

=====

The following are some requirements for callback functions in multiple instance applications when compiling with Microsoft C version 7.0 or C version 8.0 for protected-mode-only Win16:

- Callback functions must be branded with the "__export" keyword. The compiler switches -GA -GEf force all far functions in a module to be __export. The optimum method is to add the __export keyword to the particular callback function and not use the -GEf switch.
Example:

```
    BOOL CALLBACK __export CallBackProc
```

- Callback functions called via a MakeProcInstance must load DS from the value in AX. Use the compiler switch -GA with the -GEa switch to generate prolog code to load DS from the value in AX on all __export functions. The default for -GA is load DS from SS, which is not a valid assumption in callbacks. The code when loaded into memory should have three NOPs instructions at __export function entry points. Check with your favorite debugger that can show mixed/asm view once the module is loaded into memory.
- Callback functions must have an export record in the EXE header. The classic method is to list the function in the EXPORTS section of the module's .DEF file. The callback's module can be built with the - GA -GEe switches to place an export record in the .OBJ at compile time for all __export functions.
- Callback functions must assume SS != DS, DS not loaded on function entry. Compile with -Aw memory model customizer to have correct code generated.
- Callback functions must not call C run-time library code contained in the application (xLIBCyW.LIB) libraries. They assume SS == DS.

Additional reference words: 7.00 8.00 3.10 MakeProcInstance DLL

KBCategory:

KBSubcategory: KrTskinsMpi

INF: Passing Modified Environments to Child Processes

Article ID: Q102958

The information in this article applies to:

- Microsoft Windows Software Development Kit (SDK) for Windows, version 3.1
-

SUMMARY

=====

To provide a new environment for Windows applications, the LoadModule() function must be used. LoadModule() is similar to the MS-DOS Interrupt 21h Function 4Bh call, which uses a structure to find the new environment.

The element "envseg" must be set to the selector of a memory block that contains a correctly formatted environment block. The loader makes a copy of this memory block for the child process inside LoadModule(). This copy is formatted similar to an MS-DOS 2.x environment block. It does not have the MS-DOS versions 3.x and later additional information, such as the full pathname of the task, attached to the end. If startup code (such as the Microsoft C run time earlier than C version 7.0) looks for this additional information, a general protection (GP) fault will occur. If you are certain that the task started via LoadModule() does not make this assumption, then it is safe to use it.

Another bug with LoadModule() is the ownership of the copied memory created by the loader. It's set to that of the parent, and therefore it needs to outlive the child or its environment block will be free'd when the parent terminates.

If it's necessary to provide a modified environment to DOS applications, the supported technique is to use an MS-DOS batch file. The batch file first sets the new environment settings and then starts the DOS application.

```
SET FOOS=BALL
DOSAPP
```

If the application being started is not a Windows application, the .BAT file technique is the only supportable method. Regarding using LoadModule(), below is sample code that passes the current task's environment as envseg:

Sample Code

```
typedef struct tagCMDSHOW
{
    WORD wFirst;
    WORD wSecond;
}
```

```

CMDSHOW;

typedef CMDSHOW FAR * LPCMDSHOW ;

typedef struct tagPARAMETERBLOCK
{
    WORD        wEnvSeg;
    LPSTR       lpCmdLine;
    LPCMDSHOW  lpCmdShow;
    DWORD      dwUnused;
}
PARAMETERBLOCK;

typedef PARAMETERBLOCK FAR * LPPARAMETERBLOCK ;

CMDSHOW        CmdShow;
PARAMETERBLOCK ParameterBlock;
char           szCmdName[] = "TASKMAN.EXE";
char           szCmdLine[] = "";

ParameterBlock.wEnvSeg          = HIWORD(GetDosEnvironment());
ParameterBlock.lpCmdLine        = szCmdLine;
ParameterBlock.lpCmdShow        = &CmdShow;
ParameterBlock.lpCmdShow->wFirst = 2;
ParameterBlock.lpCmdShow->wSecond = SW_SHOW;
ParameterBlock.dwUnused         = NULL;

LoadModule(szCmdName, &ParameterBlock);

```

Additional reference words: 3.10 spawn process environment INT

KBCategory:

KBSubcategory: KrTskinsSpawn

INF: Spawn an Application and Wait Sample Code

Article ID: Q80226

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

ARNIE.ZIP is a file in the Software/Data Library that demonstrates how an application designed for the Windows environment can spawn another application and suspend its execution until the spawned application terminates. The sample application demonstrates detecting when a task starts and completes using the TOOLHELP.DLL dynamic-link library, a new feature of Windows 3.1.

ARNIE can be found in the Software/Data Library by searching on the word ARNIE, the Q number of this article, or S13266. ARNIE was archived using the PKware file-compression utility.

More Information:

The TOOLHELP DLL supplies a notification mechanism by which an application can be notified of various events that occur in the system. Two such events are task startup and task termination. To install a notification hook, the application calls the NotifyRegister function, specifying a callback procedure. The callback procedure is called when specified events occur in the system. To determine when an application starts and ends, watch for the NFY_TASKSTART and NFY_TASKEND notifications in the callback procedure. When the application no longer requires notifications, call the NotifyUnRegister function.

Additional reference words: 3.10 softlib ARNIE.ZIP

KBCategory:

KBSubcategory: KrThTaskman

INF: Using TOOLHELP to Determine Free System Resources

Article ID: Q76247

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

In Windows version 3.1, a Windows application can obtain the amount of free system resources in two ways:

1. By calling the GetFreeSystemResources function, which is exported by KERNEL
2. By calling the SystemHeapInfo function, which is exported by the TOOLHELP.DLL dynamic-link library

Under Windows 3.0, the GetFreeSystemResources function is not available. If TOOLHELP.DLL is present, a Windows 3.0 application can use SystemHeapInfo. If TOOLHELP.DLL is not present, there is no supported method for a Windows 3.0 application to obtain the amount of free system resources.

More Information:

The second prerelease of Windows 3.1 does not implement the GetFreeSystemResources function. This function should be implemented before the third prerelease is shipped.

The version of TOOLHELP.DLL included with the second prerelease of Windows 3.1 does not implement the SystemHeapInfo function. This function should be implemented before the third prerelease is shipped.

The version of TOOLHELP.DLL included with the second prerelease does contain two functions for obtaining the current amount of free system resources: UserHeapInfo and GDIHeapInfo. The THSAMPLE sample application included with the second prerelease of the Windows Software Development Kit version 3.1 shows how to use these calls. In the third prerelease, these two functions will be replaced by the single function, SystemHeapInfo.

Additional reference words: 3.00 3.10 pre-release

KBCategory:

KBSubcategory: KrThSysinfo

INF: Retrieving Application Exit Code in MS-DOS Window
Article ID: Q83301

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

Under Windows version 3.1, the value returned by an MS-DOS application is available to other applications in the system. Under Windows version 3.0, however, this value is not available; the exit code returned is always zero.

More Information:

Windows supports an MS-DOS window in which to run MS-DOS (non-Windows) applications. The WINOLDAP module serves as an interface for the application and the remainder of the Windows environment. When the MS-DOS application terminates, WINOLDAP retrieves the application's exit code. Then WINOLDAP itself terminates, using the retrieved exit code as its own exit code.

An application developed for the Windows environment can retrieve the exit code from WINOLDAP. The application defines a call-back function and calls the NotifyRegister function in the tool helper library (TOOLHELP.DLL). The call-back function will receive an NFY_EXITTASK message containing the exit code each time a task terminates.

The Windows 3.1 WINOLDAP module can return a number of error codes which are listed and explained in the "Microsoft Windows Software Development Kit Programmer's Reference Volume 1," page 277.

Additional reference words: 3.10

KBCategory:

KBSubcategory: KrThTaskman

INF: Sample Windows Application Produces Stack Trace

Article ID: Q92537

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

STKTRACE is a sample Windows application that contains the complete code to obtain a stack trace of the "current task." The stack trace includes symbolic information extracted from the symbol files, similar to that produced by the Windows debug kernel at the time of a FatalExit. In this sample, the stack trace is logged into a buffer and output to a message box.

The STKTRACE sample uses the tool helper library and can be used in the Windows versions 3.0 and 3.1 environments because TOOLHELP.DLL is a redistributable.

STKTRACE is available in the Software/Data Library and can be found by searching on the word STKTRACE, the Q number of this article, or S13721. STKTRACE was archived using the PKware file-compression utility.

More Information:

Usually, the Windows debug kernel produces a stack trace on the debug terminal when the SPACE BAR or ENTER key is pressed at the time of a FatalExit. For more information on stack traces produced by the debug kernel, query on the following words in the Microsoft Knowledge Base:

stack trace space bar

However, when you are writing a debugger or a large complex application, it might be necessary to produce stack traces in your application. Then, the STKTRACE sample code can be helpful.

This sample consists of two main modules, GetTrace.c and GetSymbol.c, and a driver module, StkTrace.c, which merely calls the API from the GetTrace.c module. The GetTrace.c module contains code to walk the stack of the current task by using the tool helper library and the GetSymbol.c module contains code to obtain symbol names from corresponding symbol (.SYM) files by using the symbol file format. These two modules can be plugged into any application or a DLL (dynamic-link library).

Note that the THSAMPLE application in the \SDK31\SAMPLES\TOOLHELP directory also illustrates how to produce stack traces for a given task that is not the current task. In contrast, the STKTRACE sample walks the stack of the current task by obtaining the register values from the stack and using the StackTraceCSIPFirst() and StackTraceNext() APIs from the tool helper library. Also, the STKTRACE sample provides symbolic information in the

stack trace.

The GetSym.c module provides symbolic information by using the symbol file format documented in the Microsoft Windows SDK version 3.1 "Programmer's Reference, Volume 4: Resources" manual. Given a segment:offset address, this module finds the "nearest" public symbol in the corresponding symbol (.SYM) file.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrThUsedebug

PRB: Error in the THSAMPLE Sample Application

Article ID: Q99335

Summary:

The Microsoft Windows version 3.1 Software Development Kit (SDK) provides a sample called THSAMPLE, which is in the SDK subdirectory provide the type of information that HEAPWALK shows.

SYMPTOMS

When the THSAMPLE sample application is run and the Enable List option is enabled from the Notification menu, some of the notification messages are not seen in the notification window.

This problem can be demonstrated, for example, by starting the Windows Help application when the Enable List option is enabled in the THSAMPLE application. (Note that it is preferable to turn off the task switch notifications using the Notification/Task Switch Filter option).

CAUSE

There is a simple error in the THSAMPLE application. THSAMPLE uses NotifyRegister() to register a callback function that receives all the notification messages. This callback function is then posting a user-defined message to itself using PostMessage() for every notification message it receives, and displaying it in the appropriate window.

There is a default limit, however, of eight messages that an application's message queue can hold, whereas the callback could receive a lot more notifications (for example, when some applications are started, a number of LOADSEG notifications are sent).

RESOLUTION

To correct this behavior, the application's message queue size must be increased by using SetMessageQueue() set to an appropriate number. Note that starting and exiting some applications such as Windows Help, Word for Windows, or Excel will produce a large number of LOADSEG and FREESEG notifications. Therefore, it is better to set the message queue size to the maximum of 120 in order to be able to look at most of the notifications.

Additional reference words: 3.1 3.0 problem winhelp

KBCategory:

KBSubcategory: KrThUseDebug

INF: Chaining NotifyRegister Callbacks Issuing Notifications

Article ID: Q99671

Summary:

Microsoft Windows version 3.1 can register callback functions with the NotifyRegister application programming interface (API) function, which is called on notifications. Callback functions that result in a notification being issued, however, themselves fail to chain on to other registered callbacks. This causes negative side-effects to other applications that use the NotifyRegister function.

More Information:

NotifyRegister can install callback functions that are called in response to events such as the starting and ending of an application, the loading of a segment, the freeing of a module, and so forth. Only one callback can be installed per task running in Windows. When more than one task is running, each can install a notification callback. Multiple callback functions are called by a chaining mechanism implemented in TOOLHELP when an event occurs. Callback functions are called in the order they were installed by the tasks. See the Windows Software Development Kit (SDK) version 3.1 "Programmer's Reference, Volume 2: Functions" manual for more information on the NotifyRegister function.

If a callback routine performs an operation that causes another notification to be issued, then the callback functions are not called in the usual chained manner. The problem is not that the callback routine isn't handling reentrancy; rather, NotifyRegister has not been implemented to handle callbacks that result in notifications being issued.

For example, a common procedure is to call OutputDebugString to help debug an application. However, callback functions installed by the NotifyRegister function cannot use OutputDebugString because this function call causes the NFY_OUTSTR notification to be issued. When OutputDebugString returns, TOOLHELP can't chain on to the next installed callback function. The result is that only the first callback is called for the event at which OutputDebugString was called.

If printing debug messages from the callback routine is desired, call PostMessage to inform the application of the event, and call OutputDebugString from the application's window procedure instead of calling it directly from the callback routine. Using PostMessage is necessary for the application to perform any operation, not just a call to OutputDebugString, which causes a notification event to occur. Incidentally, the documentation states that "the notification callback function cannot use any Windows function, with the exception of Tool Helper functions and PostMessage."

Having this type of error in an installed NotifyRegister callback results in negative side-effects in other applications that use the NotifyRegister function. Microsoft Visual C++ version 1.0 uses TOOLHELP notifications as part of the integrated development environment debugger. If any application is running that has installed

a callback that causes notifications to be issued, calling OutputDebugString for example, Visual C++ will not be able to initiate debugging an application. Upon starting to debug, Visual C++ displays an error message in a dialog box stating:

```
DEBUG ERROR: Could not load debuggee.  Unknown Error in  
Windows (-22)
```

Once the application uninstalls the errant callback routine, Visual C++ can debug without this error.

Additional reference words: msvc s_c c8 8.00

KBCategory:

KBSubcategory: KrThInterrupts

INF: Sample Windows Application to Unload DLLs from Memory
Article ID: Q96312

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1
-

Summary:

During development, dynamic-link libraries (DLLs) can sometimes be left in memory with no applications using them. To unload DLLs that have been left in memory, developers must exit and then restart Windows, which can be very inconvenient. DLL UNLOADER is a sample Windows application that lets developers select a DLL, show information about it, and unload it from the system if desired; this eliminates the need to restart Windows.

UNLOADER can be found in the Software/Data Library by searching on the word UNLOADER, the Q number of this article, or S14120. UNLOADER was archived using the PKware file-compression utility.

More Information:

There are two ways that a DLL can be left in memory after all applications that use it have exited:

- The application loads a DLL by calling LoadLibrary and doesn't call FreeLibrary to unload it.
- The application causes a general protection (GP) fault and is terminated by Windows.

Either of these occurrences can cause difficulties for developers. For example:

If a DLL is left in memory with no application using it, and is then recompiled and executed, the new version of the DLL will not be loaded because Windows thinks the DLL is already loaded (because the module names are the same). If you try to debug the DLL with CodeView for Windows, the source code window will display the latest version's source code; however, the old version's code, which is still in memory, will be executed. This version mismatch causes CodeView to appear to not be working properly. To resolve this problem, you must either exit and restart Windows or force Windows to unload the old version of the DLL.

The DLL UNLOADER sample calls ToolHelp to obtain the list of modules currently loaded in the system. Because this list contains modules that belong to both DLLs and applications (which are tasks), DLL UNLOADER filters out the modules that belong to tasks so that they cannot be unloaded accidentally and cause the system to crash. The list box stores the module name and handle for each module that doesn't belong to a task. When the user selects a module, its handle

is used to obtain information about it and/or unload it.

To unload a DLL, DLL UNLOADER first calls GetModuleUsage to retrieve its usage count and then repeatedly calls FreeLibrary until the usage count drops to zero; then Windows unloads the DLL. To obtain information about a DLL, DLL UNLOADER calls ModuleFindHandle and displays the information in a dialog box.

Finally, DLL UNLOADER creates an .INI file in the directory where its executable file resides to remember its last position on the screen before it exits.

Additional reference words: 3.10 CVW TOOLHELP UNLOADER

KBCategory:

KBSubcategory: KxDllUnloadfail KrThModules

INF: MS-DOS Application Characteristics Under Windows

Article ID: Q73668

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

This article discusses the mechanism that the Microsoft Windows graphical environment uses to run an MS-DOS (non-Windows) application.

More Information:

In the Window environment, each MS-DOS application runs has a Windows application to act as its agent. When the MS-DOS application is running in a window under enhanced mode Windows, and the MS-DOS application makes a call to the display hardware, the agent intercepts the call and places the character into its window. To the MS-DOS application, the agent acts as a virtual copy of the display hardware.

Note: The Windows agent does not manage the display; it simply renders the MS-DOS application's display into a window.

To determine if an application is a MS-DOS application, check the application's name to see if it matches the name of the MS-DOS application agent. The module name of the MS-DOS application agent is WINOLDAP. The following code fragment performs this check:

```
BOOL IsThisWOAWindow(HWND hWnd)
{
    BOOL IsWOA;
    HANDLE hModWOA;

    IsWOA = FALSE;
    if (hModWOA = GetModuleHandle("WINOLDAP"))
    {
        if (hModWOA == (HANDLE)(GetClassWord(hWnd, GCW_HMODULE)))
        {
            IsWOA = TRUE;
        }
    }
    return IsWOA;
}
```

To determine how many MS-DOS applications are running at any given time, call the code above from a loop that enumerates the handles of all windows in the system.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrWoaMonitoring

FIX: Program Execution Halted Until Key Press

Article ID: Q70799

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

PROBLEM ID: WIN9103007

SYMPTOMS

In a Windows MS-DOS window, when a command is invoked and has its input redirected to come from a text file, the command does not proceed until the user presses a key on the keyboard.

STATUS

Microsoft has confirmed this to be a problem in Windows version 3.0. This problem was corrected in Windows version 3.1.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrWoaMisc

PRB: Activating Full-Screen DOS App from Icon in Enhanced Mode
Article ID: Q69895

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

SYMPTOMS

In Microsoft Windows enhanced mode, when an application calls the `OpenIcon` or `ShowWindows` functions to activate a full-screen MS-DOS application from an icon, the activation fails. These functions perform as documented in real and standard mode.

RESOLUTION

To avoid this problem, simulate the result of an ALT+TAB key combination on the MS-DOS application icon: Send a `WM_ACTIVATE` message to the MS-DOS application, and then specify its handle in a call to `ShowWindow` function. For example:

```
SendMessage(hDOSApp, WM_ACTIVATE, 1, MAKELONG(hDOSApp, TRUE));  
ShowWindow(hDOSApp, SW_SHOWNORMAL);
```

When the application sends the `WM_ACTIVATE` message, `wParam` is set to 1 because the application is being invoked by simulating the keyboard. The high-order word of `lParam` is set to `TRUE` (non-zero) because the MS-DOS application is represented by an icon.

The method works in all three Windows modes for windowed and full-screen MS-DOS applications.

Additional reference words: 3.00 MICS3 R1.3

KBCategory:

KBSubcategory: KrWoaMonitoring

INF: Calculating Memory Requirements for DOS Applications
Article ID: Q43041

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.1 and 3.0
-

Summary:

It is extremely difficult for an application to determine in advance how much memory an MS-DOS (non-Windows) application will require to run. This data can be estimated under real and standard modes; however, there is no method to determine this information in enhanced mode.

More Information:

The Windows module WinOldAp is required to run MS-DOS applications. One complication is that once WinOldAp is in memory, it can run many applications. In the following table, "PIF Req'd" indicates the amount of memory required for the application as listed in the corresponding PIF file and "PIF Des'd" indicates the amount of memory desired by the application as listed in the corresponding PIF file. The following table indicates the difference between applications:

PIF Req'd	PIF Des'd	Free Mem Before	Free Mem After	Memory Usage	
160	160	418	155	263	First copy run
160	160	153	152	1	Second copy run

As this table indicates, the second copy of the application costs almost no memory. This is because the two copies of the application share the same memory and are swapped in and out of memory (to disk, to a RAM drive, or to expanded memory, depending on the WIN.INI settings).

Another factor is the order in which MS-DOS applications are loaded. The statistics below demonstrate these differences. In the first case, an 80K program is loaded followed by a 160K program. In the second case, the 160K program is loaded followed by the 80K program:

Case 1:

PIF Req'd	PIF Des'd	Free Mem Before	Free Mem After	Memory Usage	
80	80	418	244	174	App #1 runs first
160	160	244	37	207	App #2 runs second
				381	Total memory usage

Case 2:

PIF Req'd	PIF Des'd	Free Mem Before	Free Mem After	Memory Usage	
-----	-----	-----	-----	-----	
160	160	418	154	264	App #2 runs first
80	80	154	153	1	App #1 runs second
				265	Total memory usage

These results are not as unusual as they may appear. In the second case, the larger application (App #2) is loaded first. This sets the WinOldAp swapping partition large enough to hold the application. When the smaller application (App #1) is run, it fits into the existing partition. In contrast, in the first case, the smaller application is run first, therefore the swapping area is not set large enough to hold the larger application. When the larger application is loaded, WinOldAp must create a completely separate partition to hold it.

The amount of memory needed to run an old application varies, depending on the following:

1. Whether or not WinOldAp is loaded
2. Whether the existing swap area (if any) is large enough to hold it
3. Whether the application screen is saved in text or graphics mode
4. Whether large-frame, small-frame, or no EMS is in use
(this variable does not apply to Windows version 3.1)
5. The contents of the Memory Required and Memory Desired fields in the PIF file
6. Other factors related to the inner workings of the Windows memory manager and the WinOldAp module

These methods are not applicable to enhanced mode Windows. The only way to determine if a MS-DOS application will run under enhanced mode is to attempt to run the application and see if the attempt succeeds. Even this information is not available to another application because the WinExec return value only indicates that WinOldAp has been successfully loaded into memory. This value does not contain any information regarding the real target application.

The handling of this type of problem is addressed by Windows version 3.1. In Windows version 3.1, the TOOLHELP library can be used to retrieve the Exit Code of a Windows application. This also works in Windows version 3.0, however, the problem is that WINOLDAP (the MS-DOS application's Windows agent) for Windows version 3.0 always exits with exit code 0. Therefore, there is no way to obtain results of the attempted MS-DOS application run. The version of WINOLDAP included with Windows version 3.1 will exit with the exit code of the MS-DOS application that was run, or one of the following special values:

```
;  
; Special WINOLDAP exit codes
```



```
;  
EXIT_NoFile      EQU  81h ; Could not start due to file or  
                  ; directory access problem  
EXIT_NoMem       EQU  82h ; Could not start due to insufficient  
                  ; memory  
EXIT_Crash       EQU  83h ; VM crashed (abnormal termination)  
EXIT_BadVer      EQU  84h ; Could not start due to bad version  
EXIT_ExecFail    EQU  85h ; Could not start because MS-DOS EXEC failed  
EXIT_TaskAPIFail EQU  86h ; Could not start because task switch API  
                  ; refused start (standard mode only)
```

NOTE THAT THESE SPECIAL CODES MAY OVERLAP WITH AN EXIT CODE USED BY THE MS-DOS APPLICATION. If this happens, there is no way to correct it other than to change the MS-DOS application to use different exit codes that do not conflict with these special ones.

ALSO NOTE: The exit code of a .BAT file run is always 0. This is a property of COMMAND.COM, which is part of MS-DOS.

Additional reference words: 3.00 3.10 3.x

KBCategory:

KBSubcategory: KrWoaMisc

INF: Determining Windows Version, Mode from MS-DOS App
Article ID: Q75338

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.1

Summary:

The following assembly language code fragment can be used in an MS-DOS (non-Windows) application to determine if it is running in a Windows MS-DOS window, and if so, which version and mode of Windows is running.

test_win proc near

; check for Windows 3.1

```
mov     ax,160ah           ; WIN31CHECK
int     2fh               ; check if running under Win 3.1.
or      ax,ax
jz      RunningUnderWin31 ; can check if running in standard
                          ; or enhanced mode
```

; check for Windows 3.0 enhanced mode

```
mov     ax,1600h          ; WIN386CHECK
int     2fh
test    al,7fh
jnz     RunningUnderWin30Enh ; enhanced mode
```

; check for 3.0 WINOLDAP

```
mov     ax,4680h          ; IS_WINOLDAP_ACTIVE
int     2fh
or      ax,ax             ; running under 3.0 derivative?
jnz     NotRunningUnderWin
```

; rule out MS-DOS 5.0 task switcher

```
mov     ax,4b02h          ; detect switcher
push    bx
push    es
push    di
xor     bx,bx
mov     di,bx
mov     es,bx
int     2fh
pop     di
pop     es
pop     bx
or      ax,ax
jz      NotRunningUnderWin ; MS-DOS 5.0 task switcher found
```

```

; check for standard mode Windows 3.0

mov     ax,1605h                ; PMODE_START
int     2fh
cmp     cx,-1
jz      RunningUnderWin30Std

; check for real mode Windows 3.0

mov     ax,1606h                ; PMODE_STOP
int     2fh                    ; in case someone is counting
; Real mode Windows 3.0 is running
jmp     NotRunningUnderWin

RunningUnderWin30Std:

; Standard mode Windows 3.0 is running
jmp     NotRunningUnderWin

RunningUnderWin31:

; At this point: CX == 3 means Windows 3.1 enhanced mode
;                CX == 2 means Windows 3.1 standard mode
jmp     NotRunningUnderWin

RunningUnderWin30Enh:

; Enhanced mode Windows 3.0 is running

NotRunningUnderWin:

ret

test_win endp

```

Additional reference words: 3.10 3.1
KBCategory:
KBSubcategory: KrWoaMisc

INF: Keeping a DOS Window Active Under Standard and Real Mode
Article ID: Q75433

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0
-

Summary:

When a DOS (non-Windows) application running in a DOS window is in the process of performing "critical" activity (such as asynchronous network communication), switching away from the application could result in the loss of data or a system crash. Under standard mode and real mode Windows, a DOS application can prevent Windows from switching away from it. This scheme can also be used by an application run before Windows is started to prevent switching away from a DOS application.

To implement this method, an application hooks interrupt 6Fh. Before switching away from a DOS application, Windows makes an int 6Fh call with the AX register set to 204h. This call is made if any application has hooked int 6Fh. If the AX register contains 0 on return from the interrupt, the switch is allowed; otherwise, the switch is prevented.

This technique can cause problems if the only applications that hook int 6Fh that are running in the system do not use this convention. In that case, no switching will be allowed. To avoid this situation, the int 6Fh call can be disabled from the SYSTEM.INI file. In the [NonWindowsApp] section, a TaskSwitchInt6f=OFF entry disables the int 6Fh.

Although this method can be used by an application running under the MS-DOS 5.0 task switcher, a full-fledged application programming interface related to switching has been implemented. These functions would be much more useful for applications.

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrWoaMonitoring

INF: Determining What Mode and Version of Windows Is Running
Article ID: Q75943

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

An MS-DOS (non-Windows) application can determine whether it is running within an MS-DOS session under Windows. This article provides an assembly language program that demonstrates how this can be accomplished.

More Information:

The following code determines whether Windows is running; if Windows is running, the code determines what mode Windows is in and what version of Windows is running.

Sample Code

```
.MODEL LARGE
.CODE
        public  _iswin
;
;      Procedure to determine if in Windows or not.
;      (From "Microsoft Systems Journal," March 1991, page 113)
;      0:      Windows not running
;      1:      Windows/386 2.x is running
;      3:      Windows 3.x is in 386 enhanced mode
;      4:      Windows 4.x is in 386 enhanced mode
;      127:    Windows/386 2.x is running
;      128:    Windows 3.0 is in real mode
;      255:    Windows 3.0 is in standard mode
;
_iswin  PROC    FAR

        MOV     AX,4680H           ;Is Windows 3.0 running?
        INT     2FH
        XOR     AL,80H
        MOV     CL,AL
        MOV     AX,1600H          ;Is Windows 3.x 386 enhanced mode
        INT     2FH              ;running?
        AND     AL,7FH
        OR      AL,CL
        CMP     AL,80H
        JZ      T1                ;Not Windows enhanced mode.

; Windows enhanced mode is running. At this point,
; AL contains the major version number of Windows
; and AH contains the minor version number.
; However, for this example, only the major version
```

```

; number is used.
JMP      GOHOME          ;Windows Enhanced, Go Home

T1:      MOV      AX,1605H      ;Simulate Initialization.
        XOR      BX,BX
        MOV      ES,BX
        XOR      SI,SI
        MOV      DS,SI
        XOR      CX,CX
        MOV      DX,0001H      ;MS-DOS extender
        INT      2FH           ;Windows real or Windows standard
        CMP      CX,+00
        JNZ      T2

        MOV      AX,1606H      ;Simulate exit
        INT      2FH

T2:      MOV      AL,80H
        OR       AL,CL

GOHOME:  MOV      AH,4CH
        INT      21H

_iswin   ENDP
        END

```

Additional reference words: 3.0

KBCategory:

KBSubcategory: KrWoaMisc

INF: Full-Screen DOS Apps Slow Timer Messages in Enhanced Mode
Article ID: Q76390

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows versions 3.0 and 3.1
-

Summary:

In 386 enhanced mode, Windows system timer (WM_TIMER) messages are generated less often when a DOS application runs full-screen with its PIF file specifying background execution than when the same DOS application runs in a window.

When a full-screen DOS application is running, Windows does not need to track the mouse and update the screen. Therefore, Windows runs with a low priority in the background. Windows applications receive fewer WM_TIMER messages because the number of timer ticks that Windows receives decreases from 18.2 per second to 2 per second or even fewer.

More Information:

Windows's 386 enhanced mode architecture and display mechanism account for the way that WM_TIMER messages behave when DOS applications are running.

First of all, WM_TIMER messages are not guaranteed to be sent exactly at a specified time; they may be sent any time after the specified time has elapsed. For example, a WM_TIMER message set for 60 milliseconds may be sent at exactly 60 milliseconds, or it may be sent more than two minutes later. WM_TIMER messages behave this way because they have the lowest priority of any Windows message. Applications receive WM_TIMER messages only after all other messages in the system have been processed (that is, when all message queues are empty).

In 386 enhanced mode, Windows uses a virtual machine architecture that sets up a virtual machine (VM) for each DOS application. Windows is essentially a DOS application and creates a VM for itself called the System VM. The System VM runs Windows and all Windows applications; each DOS application receives a separate VM.

Virtual machines act similar to 8086 processors with additional privileged instructions provided by the 80386 processor's virtual-8086 mode. Virtual-8086 mode differs from real mode in that it has memory protection, virtual memory, and privilege-checking mechanisms. Each virtual machine also has an optional protected-mode portion. The System VM uses its protected mode portion to run Windows applications.

The Windows Virtual Machine Manager (VMM) controls the multitasking of the virtual machines; it manages memory, CPU execution time, and device coordination. The VMM runs in the 80386 processor's 32-bit flat

memory model along with Windows virtual devices (VxDs), and preemptively schedules the virtual machines.

Foreground virtual machines running DOS applications always receive 18.2 timer ticks each second so that the VM looks exactly like a stand-alone 8086 machine. When the System VM is in the foreground, it (and Windows) receives 18.2 timer ticks per second as well.

Background virtual machines do not receive 18.2 timer ticks per second; they receive 2 or fewer ticks per second because the VMM reduces their relative priorities. As a result, the foreground VM can run faster.

When the System VM is in the background and a full-screen DOS application is in the foreground, Windows receives fewer timer ticks, and therefore runs slower than normal. Windows applications process their messages slower, causing WM_TIMER messages to be delayed.

Windowed DOS applications behave differently from full-screen DOS applications. Windows creates a special agent application, WINOLDAP, to run windowed DOS applications. WINOLDAP's job is to place all output from a DOS application into the client area of a window. Windows controls all parts of the display, including the area of the windowed DOS application, and tracks the mouse pointer.

Under such circumstances, Windows is running in the background with a high priority. The windowed DOS application is running in the foreground in a VM. The VMM preemptively multitasks the VMs so that the DOS application can run and Windows can manage the screen. Because the System VM has a high priority, it receives more timer ticks per second than other background VMs. Windows and Windows applications, therefore, process more messages, which allows WM_TIMER messages to be sent more often.

Even though background VMs receive fewer timer ticks per second, they still receive time slices according to their relative priorities.

Note: This article applies to DOS applications that have a PIF file set for background execution. If the DOS application is running full-screen in the foreground with its PIF file set to exclusive processing, all other applications, including Windows, will be suspended, just as if the DOS application were run under Windows in standard mode. If the DOS application runs in a window with exclusive processing set, then Windows does not get suspended because it must manage the screen output for the DOS application.

Additional reference words: 3.00 3.10

KBCategory:

KBSubcategory: KrWoaMisc

INF: Requested Contents for Windows Problem Reports

Article ID: Q51503

The information in this article applies to:

- Microsoft Windows Software Development Kit for Windows version 3.0

Summary:

This article describes Microsoft's preferred format for submitting problem reports on Windows. Using this format will help us to understand the precise context of your problem report, enabling us to provide better answers to you and more helpful problem reports to our development staff.

More Information:

Your problem report should be structured as follows:

SEVERITY:
ENVIRONMENT:
DESCRIPTION:
CONFIGURATION:
 Hardware:
 Drivers:
 Software:
DEBUG INFORMATION:

Each of these sections is described below:

SEVERITY: 1, 2, 3, or 4. Use the following guidelines for severity:

- 1 Critical -- Software does not work at all (crashes) or causes loss/corruption of data. Any situation where the system hangs or requires a cold or warm boot.
- 2 Major -- A feature/function does not operate as designed. Any command or function that produces incorrect results or renders a portion of the product unusable.
- 3 Minor -- Problems are generally minor in nature and do not prevent program from running (spelling errors, screen display errors, and so forth), or the results are misleading and/or difficult for the user to understand.
- 4 Enhancement -- Problem is a design or feature fault that can be addressed in a future release.

ENVIRONMENT:

WINDOWS REAL or 286 PROTECT or 386 PROTECT release Version x.xx

(Include debug info line from the bottom right of screen, if

applicable)

DESCRIPTION:

Below is a description of the "ideal" problem report. The best problem reports are clear, complete, and concise. These are some points that will make it easier to track problem reports and enter them in our problem database.

The report should state the steps necessary to reproduce the problem, the results, and (if available) a summary of the debug output. Try to include enough information that we can reproduce the problem here, but also try to keep the reports concise.

The configuration should be at the end of the problem report so that people reading the report can get the gist of the problem without having to wade through information that may not be relevant (but the information is there in case it is relevant).

CONFIGURATION:

Hardware: CPU/Memory/Monitor/keyboard/printer/modem/etc.

Drivers: Installed device drivers

Software: TSRs/Network drivers/etc.

DEBUG INFORMATION:

The following is a sample of debug output that can be helpful in locating a problem:

GENERAL PROTECTION VIOLATION

CS=0EED SS=0F35 DS=0F35 ES=0000 FS=0000 GS=0000

-- NV UP EI PL NZ NA PE NC

0EED:00000387 MOV AL, BYTE PTR AL, BYTE PTR [BX]

DS:23D0=INV:0003

ln

No symbols found

.dg cs

004D: 0063p CODE NOTEPAD (0EEE) 1065,103D

103D: 01FFp CODE USER (05BE) 104D,102D

1165: 0061p CODE GDI (03AE) 101D,100D

115D: 001Bp FREE 009D,0075

07B5: 000Bp CODE KEYBOARD (018E) 035D,0FE5

Additional reference words: 3.00

KBCategory:

KBSubcategory: KrWoaMonitoring

