

Microsoft* Win32 SDK Knowledge Base

Prepared 11/17/93



Base Topics



GDI Topics



Networking Topics



User Topics



Tools Topics



Win32s Topics
















































Sample Program Descriptions

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.



Base Topics

- [INF: Two Types of Priority Control Added](#)
- [INF: FILE_FLAG_WRITE_THROUGH and FILE_FLAG_NO_BUFFERING](#)
- [Sample: Security API Functions Demonstration](#)
- [PRB: SetConsoleOutputCP\(\) Not Functional](#)
- [Sample: Common Dialog DLL](#)
- [Sample: Determining Drive and File System Type](#)
- [Sample: Walking a Directory Tree](#)
- [PRB: ERROR_SEM_TIMEOUT Not Documented](#)
- [SAMPLE: Process API Functions Example](#)
- [Sample: Creating Resource-Only DLLs](#)
- [INF: Objects Inherited Through a CreateProcess Call](#)
- [Sample: Demonstration of Setting File Attributes](#)
- [INF: Correct Use of Try/Finally](#)
- [SAMPLE: Simple DLL Demonstration](#)
- [Sample: Virtual Memory API Function Demonstration](#)
- [INF: NT Consoles Do Not Support ANSI Escape Sequences](#)
- [INF: Setting the Console Configuration](#)
- [Sample: CreateProcess\(\) Priority Demonstration](#)
- [Sample: Demonstration of Setting Console Text Color](#)
- [Sample: Asynchronous I/O Demonstration](#)
- [Sample: Demonstration of the Console API Functions](#)
- [INF: Replacing the Shell \(Program Manager\)](#)
- [INF: Four New Win32 Memory-Handling APIs](#)
- [INF: Secure Erasure Under Windows NT](#)
- [INF: First and Second Chance Exception Handling](#)
- [INF: Thread Local Storage Overview](#)
- [INF: CPU Quota Limits Not Enforced](#)
- [INF: Using GMEM_DDESHARE in Win32 Programming](#)
- [INF: Direct Drive Access Under Win32](#)
- [INF: Byte-Ordering in a Data Packet Under NDIS](#)
- [INF: Performing a Clear Screen \(CLS\) in a Console Application](#)
- [INF: AllocConsole\(\) Necessary to Get Valid Handles](#)
- [INF: How to Specify Shared and Nonshared Data in a DLL](#)
- [INF: Precautions When Passing Security Attributes](#)
- [INF: Physical Memory Limits Number of Processes/Threads](#)
- [INF: Security and Screen Savers](#)
- [INF: Preventing the Console from Disappearing](#)
- [INF: Detecting Windows NT from a DOS Application](#)
- [INF: Apps Should Wait to Free/Re-use WriteFileEx's Buffer](#)
- [INF: CreateFile\(\) Using CONOUT\\$ or CONIN\\$](#)
- [INF: FlushViewOfFile\(\) on Remote Files](#)

-  [INF: No Way to Cancel Overlapped I/O](#)
-  [INF: Restriction on Named-Pipe Names](#)
-  [INF: Getting Real Handle to Thread/Process Requires Two Calls](#)
-  [INF: Exporting Data from a DLL](#)
-  [INF: Time Stamps Under the FAT File System](#)
-  [INF: Dynamic Loading of DLLs Under Windows NT](#)
-  [INF: Implementing a](#)
-  [PRB: try/finally with Abort\(\) in try Body](#)
-  [INF: LIM-EMS Memory Limitations in a Windows NT VDM](#)
-  [INF: Examining the dwOemId Value](#)
-  [INF: Win32 Priority Class Mechanism and the START Command](#)
-  [INF: Creating Windows in Threads](#)
-  [INF: SHARE.EXE Functionality Built into Windows NT](#)
-  [PRB: try/finally with return in finally Body Preempts Unwind](#)
-  [INF: Initiating an Unwind in an Exception Handler](#)
-  [INF: Using volatile to Prevent Optimization of try/finally](#)
-  [INF: Icons for Console Applications](#)
-  [PRB: Maximum Memory Handles](#)
-  [INF: PAGE_READONLY May Be Used as Discardable Memory](#)
-  [PRB: Return Values of Performance APIs](#)
-  [INF: Sharing Win32 Services](#)
-  [INF: Determining Whether Windows NT Is Running](#)
-  [INF: Interrupting Threads in Critical Sections](#)
-  [INF: New DLL: LOCALMON.DLL](#)
-  [INF: Changes to DLL Makefiles Made for Final Release](#)
-  [INF: Impersonation Provided by ImpersonateNamedPipeClient\(\)](#)
-  [INF: Distributed Computing Environment \(DCE\) Compliance](#)
-  [INF: Process Will Not Terminate Unless System Is In User-mode](#)
-  [INF: Non-Address Range in Address Space](#)
-  [INF: Alternatives to Using GetProcAddress With LoadLibrary](#)
-  [INF: Gaining Access to ACLs](#)
-  [INF: Maximum GlobalAddAtom\(\) String Size Is 32K Characters](#)
-  [INF: Administrator Access to Files](#)
-  [INF: Passing Security Information to SetFileSecurity\(\)](#)
-  [INF: Extracting the SID from an ACE](#)
-  [INF: How to Add an Access-Allowed ACE to a File](#)
-  [INF: Computing the Size of a New ACL](#)
-  [PRB: Determining Whether App Is Running as Service or .EXE](#)
-  [INF: VirtualLock\(\) Only Locks Pages into Working Set](#)
-  [INF: Trapping Floating-Point Exceptions Under Windows NT](#)
-  [INF: FormatMessage\(\) Converts GetLastError\(\) Codes](#)
-  [INF: Validating User Accounts \(Impersonation\)](#)
-  [INF: Types of File I/O Under Win32](#)
-  [INF: FILE_READ_EA and FILE_WRITE_EA Specific Types](#)
-  [INF: Chaining Parent PSP Environment Variables](#)

-  [INF: System GENERIC_MAPPING Structures](#)
-  [INF: Default Stack in Win32 Applications](#)
-  [PRB: Code in DLL Causes Access Violation C0000005](#)
-  [INF: Starting and Terminating 16-Bit Windows Applications](#)
-  [INF: Why LoadLibraryEx\(\) Returns an HINSTANCE](#)
-  [INF: CTRL+C Exception Handling Under WinDbg](#)
-  [PRB: New Parameter for the CreateService\(\) API](#)
-  [INF: The Use of the SetLastErrorEx\(\) API](#)
-  [INF: Passing a Pointer to a Member Function to the Win32 API](#)
-  [INF: File Manager Passes Short Filename as Parameter](#)
-  [INF: Windows NT Virtual Memory Manager Uses FIFO](#)
-  [INF: Determining Memory Usage Under Windows NT](#)
-  [INF: Getting the Net Time on a Domain](#)
-  [INF: Noncontinuable Exceptions](#)
-  [INF: Validating User Account Passwords Under Windows NT](#)
-  [PRB: Unexpected Result of SetFilePointer\(\) with Devices](#)
-  [INF: Limit on the Number of Bytes Written Asynchronously](#)
-  [INF: Setting File Permissions](#)
-  [INF: Detecting Closure of Command Window from a Console App](#)
-  [INF: Definition of a Protected Server](#)
-  [INF: SetTimer\(\) Should Not Be Used in Console Applications](#)
-  [INF: Security Attributes on Named Pipes](#)
-  [INF: Using Temporary File Can Improve Application Performance](#)
-  [INF: Calling a Win32 DLL from a Win16 Application Under WOW](#)
-  [INF: Dynamically Growing Named File Mappings](#)
-  [INF: How Keyboard Data Gets Translated](#)
-  [INF: Monitoring a Log File for an Event](#)
-  [BUG: Redirecting Output to an MS-DOS Application](#)
-  [INF: SetErrorMode\(\) Is Inherited](#)
-  [INF: Calling CRT Output Routines from a GUI Application](#)
-  [INF: Getting and Using a Handle to a Directory](#)
-  [INF: The Use of PAGE_WRITECOPY](#)
-  [BUG: Problems with Local/Global Memory Management APIs](#)
-  [BUG: AllocConsole\(\) Does Not Set Error Code on Failure](#)
-  [INF: Critical Sections Versus Mutexes](#)
-  [PRB: GetPrivateProfileSection\(\) Can Read Only 32K Sections](#)
-  [INF: Using NTFS Alternate Data Streams](#)
-  [INF: RegSaveKey\(\) Requires SeBackupPrivilege](#)
-  [INF: Identifying a Previous Instance of an Application](#)



GDI Topics



Networking Topics



User Topics



Tools Topics



Win32s Topics



Sample Program Descriptions

 [Win32 SDK Knowledge Base](#)



Base Topics



GDI Topics

-  [Sample: Saving/Loading Bitmaps in .DIB Format on MIPS](#)
 -  [Sample: World Coordinate Transform](#)
 -  [Sample: AngleArc Demonstration Program](#)
 -  [Sample: Using GetDIBits\(\) for Retrieving Bitmap Information](#)
 -  [Sample: Demonstration of Using System Info API](#)
 -  [Sample: StretchBlt Demonstration](#)
 -  [Sample: Using Region-Related API Functions](#)
 -  [Sample: PlgBlt Demonstration](#)
 -  [SAMPLE: Using Graphic Paths Demonstration](#)
 -  [SAMPLE: PolyBezier\(\) Demonstration](#)
-  [Sample: GetDeviceCaps\(\) Demonstration Program](#)
 -  [Sample: PolyDraw Function Demonstration](#)
 -  [INF: Use 16-Bit .FON Files for Cross-Platform Compatibility](#)
 -  [Sample: MaskBlt Function Demonstration](#)
 -  [INF: Device Contexts: Using Across Threads](#)
 -  [INF: Transparent Blts in Windows NT](#)
 -  [INF: 16 and 32 Bits-Per-Pel Bitmap Formats](#)
 -  [INF: PSTR's in OUTLINETEXMETRIC Structure](#)
 -  [INF: Advantages of Device-Dependent Bitmaps](#)
-  [INF: Set/ModifyWorldTransform\(\) Requires SetGraphicsMode\(\)](#)
 -  [PRB: IsGdiObject\(\) Is Not a Part of the Win32 API](#)
 -  [INF: Use of DocumentProperties\(\) vs. ExtDeviceMode\(\)](#)
 -  [INF: Font-Related APIs & Structures Removed from Win32/NT](#)
 -  [INF: DEVMODE and dmSpecVersion](#)
 -  [INF: Tracking Brush Origins in Windows NT](#)
 -  [INF: Calculating the TrueType Checksum](#)
 -  [INF: Creating a Font for Use with the Console](#)
-  [INF: Creating a Logical Font with a Nonzero IfOrientation](#)



Networking Topics



User Topics



Tools Topics



Win32s Topics



Sample Program Descriptions

 [Win32 SDK Knowledge Base](#)



Base Topics




GDI Topics



Networking Topics

 [INF: Windows Socket API Specification Version 1.1](#)

 [INF: Writing a Telnet Client](#)

 [INF: Supported Versions of Windows Sockets](#)

 [INF: Using RPC Callback Functions](#)

 [PRB: RPC Installation Problem](#)



User Topics



Tools Topics



Win32s Topics



Sample Program Descriptions

 [Win32 SDK Knowledge Base](#)



Base Topics




GDI Topics




Networking Topics



User Topics

-  [PRB: AttachThreadInput\(\) Resets Keyboard State](#)
 -  [INF: Global Classes in Win32](#)
 -  [Sample: Common Dialog DLL](#)
 -  [SAMPLE: Standard DLL & Ex. of Creating a Custom Control Class](#)
 -  [INF: DDEML Application-Instance IDs Are Thread Local](#)
 -  [SAMPLE: WM_COMMNOTIFY Message is Obsolete](#)
-  [Sample: SUBCLASS Program Demonstration](#)
 -  [Sample: Communications API Function Demonstration](#)
 -  [INF: Freeing PackDDEIParam\(\) Memory](#)
 -  [INF: System Versus User Locale Identifiers](#)
 -  [INF: Multiline Edit Control Limits in Windows NT](#)
 -  [INF: Use of DLGINCLUDE in Resource Files](#)
 -  [INF: Multiple Desktops Under Windows NT](#)
 -  [INF: Clarification of COMMPROP Max?xQueue Members](#)
-  [INF: OpenComm\(\) and Related Flags Obsolete Under Win32](#)
 -  [INF: Window Message Priorities](#)
 -  [INF: Distinguishing Between Keyboard ENTER and Keypad ENTER](#)
 -  [PRB: Setting Hooks Locally or Globally](#)
 -  [INF: NULL is a Valid Return From SetWindowsHook\(\)](#)
 -  [INF: LB_GETCARETINDEX Returns 0 for Zero Entries in List Box](#)
 -  [INF: SetActiveWindow\(\) and SetForegroundWindow\(\) Clarification](#)
 -  [INF: Possible Serial Baud Rates on Various Machines](#)
 -  [INF: Memory Handles and Icons](#)
-  [INF: Debugging a System-Wide Hook](#)
 -  [INF: WM_ENTERIDLE Documentation Is Misleading](#)
 -  [INF: How to Make SPINCUBE a Global Class](#)
 -  [INF: The SBS_SIZEBOX Style](#)
 -  [SAMPLE: Control Panel Application Sample](#)
 -  [INF: Clarification of the](#)
 -  [PRB: CloseClipboard\(\) Suggests Calling DuplicateHandle\(\)](#)
 -  [INF: Differences Between hInstance on Win 3.1 and Windows NT](#)
 -  [INF: Propagating Environment Variables to the System](#)
-  [INF: 32-Bit Scroll Ranges](#)
 -  [INF: COMCTL32 APIs Unsupported in the Win32 SDK](#)
 -  [INF: Cancelling WaitCommEvent\(\) with SetCommMask\(\)](#)
 -  [INF: Win32 Shell Dynamic Data Exchange \(DDE\) Interface](#)
 -  [INF: Win32 Drag and Drop Server](#)

 [INF: ClipCursor\(\) Requires WINSTA_WRITEATTRIBUTES](#)

 [INF: Retrieving DIBs from the Clipboard](#)



[**Tools Topics**](#)



[**Win32s Topics**](#)



[**Sample Program Descriptions**](#)

 [Win32 SDK Knowledge Base](#)



Base Topics



GDI Topics



Networking Topics







































































User Topics



Tools Topics

-  [INF: NEW.H Does Not Contain new\(\) that Takes a void*](#)
-  [INF: Win32 Equivalent for C Run-Time Functions](#)
-  [Sample: Writing NTSD Extensions](#)
-  [INF: Using a Mouse with MEP Under Windows NT](#)
-  [INF: Macros to Facilitate Porting Applications to Windows NT](#)
-  [INF: Correct Use of Try/Finally](#)
-  [INF: Concatenating Resource Files Does Not Work on Windows NT](#)
-  [INF: RCDATA Begins on 32-Bit Boundary in Windows NT](#)
-  [PRB: Win32s: GetVolumeInformation Returns Incorrect Values](#)
-  [PRB: LIB.EXE: Adding Object Documentation Error](#)
-  [PRB: Problems Using COMM APIs and the DCB Structure on MIPS](#)
-  [INF: LIB32.EXE Converts Object Files to COFF Format](#)
-  [INF: Microsoft Implementation of Bit Fields in cl386 Compiler](#)
-  [INF: OS/2-to-Windows Migration Information](#)
-  [INF: Source-level Debugging Under NTSD](#)
-  [INF: Preserving Case When Assembling /Fa Listing](#)
-  [PRB: Debugging the Open Common Dialog Box in WinDbg](#)
-  [INF: Win32 Subsystem Object Cleanup](#)
-  [INF: Using MFC Build Clean Option](#)
-  [INF: Warning C4056: Overflow in Floating Point](#)
-  [INF: Unicode Conversion to Integers](#)
-  [INF: How HEAPSIZ/STACKSIZE Commit > Reserve Affects Execution](#)
-  [INF: MFC TRACE Output Not Working](#)
-  [INF: WinDbg](#)
-  [INF: Fatal Error C1001: ICE \('msc1.cpp', line 555\)](#)
-  [INF: Windows NT Compiler Always Includes chkstk\(\)](#)
-  [PRB: Windows NT: Inline Assembly Code Generation Error](#)
-  [INF: Fatal Error C1056: Out of Macro Expansion Space](#)
-  [INF: Fatal Error C1001: ICE \(file 'msc1.cpp', line 555\)](#)
-  [INF: Setting Dynamic Breakpoints in WinDbg](#)
-  [INF: Base Date for Time Differs Between NT and C/C++ 7.0](#)
-  [INF: CTYPE Macros Function Incorrectly](#)
-  [INF: Calling Conventions Supported by the 32-Bit Compiler](#)
-  [PRB: Debugging an Application Driven by MS-TEST](#)
-  [INF: Format for LANGUAGE Statement in .RES Files](#)

-  [INF: Microsoft NT C++ Is AT&T 2.1 Compatible](#)
-  [INF: Usage of the `afx_msg` Type](#)
-  [INF: Tips for Writing Multiple-Language Scripts](#)
-  [INF: Writing Multiple-Language Resources](#)
-  [INF: Using Communal Variables in MASM386](#)
-  [INF: Default Alignment of Structures and Classes](#)
-  [INF: Enabling Disk Performance Counters](#)
-  [PRB: MS-SETUP Uses \SYSTEM Rather Than \SYSTEM32](#)
-  [PRB: Selecting Overlapping Controls in Dialog Editor](#)
-  [PRB: Data Section Names Limited to Eight Characters](#)
-  [INF: Retrieving the `CMDIChildWnd` Parent Window](#)
-  [INF: MIPS Compiler Does Not Support `__inline`](#)
-  [INF: Memory Management Via `Malloc\(\)`](#)
-  [INF: Using `Cout` in an Application and DLL](#)
-  [INF: Interpreting Executable Base Addresses](#)
-  [INF: Calculating String Length in Registry](#)
-  [INF: Order of Object Initialization Across Translation Units](#)
-  [INF: Changes to `wsprintf/wvsprintf` Formatting](#)
-  [INF: `%S`, `%B`, `%C`, `printf\(\)` Format Specifier Changes](#)
-  [INF: Getting Windows NT Executable Header Information](#)
-  [INF: `WinMain\(\)` Arguments in Unicode](#)
-  [INF: Using `volatile` to Prevent Optimization of `try/finally`](#)
-  [INF: Postmortem Debugging Under Windows NT](#)
-  [INF: Use of `DLGINCLUDE` in Resource Files](#)
-  [INF: Warning 0505: No Modules Extracted from 'FILENAME.LIB'](#)
-  [PRB: GP Fault in OS/2 Subsystem](#)
-  [INF: Cross-Platform Development Under Windows NT](#)
-  [PRB: Internal Compiler Error `msc1.cpp`, Line 555](#)
-  [PRB: `LINK32.EXE\(\)`: Extended Error - File Not Found](#)
-  [INF: Using the `-ROM` Linker Switch](#)
-  [INF: Win32 `.DEF` File Usage in Applications and DLLs](#)
-  [INF: Win32 SDK Sample Build Warnings](#)
-  [INF: `LINK32` Implements New Switch: `-adjust:#`](#)
-  [INF: Size Comparison of 32-Bit and 16-Bit x86 Applications](#)
-  [INF: CTRL+C Exception Handling Under `WinDbg`](#)
-  [INF: Debugging DLLs Using `WinDbg`](#)
-  [INF: Debugging Console Apps Using Redirection](#)
-  [PRB: Building POSIX Applications Under the March Beta](#)
-  [INF:](#)
-  [INF: Watching Local Variables That Are Also Globally Declared](#)
-  [INF: Migrating Windows NT Program Groups and the Desktop](#)
-  [INF: Conforming to ANSI C Standards](#)
-  [PRB: Default Section Alignment Is 0x10000 \(64K\) by Default](#)
-  [PRB: Cannot Compile from Win32 SDK M Editor \(`MEP.EXE`\)](#)
-  [INF: `UNICODE` and `_UNICODE` Needed to Compile for Unicode](#)

-  [INF: Specifying Filenames Under the POSIX Subsystem](#)
-  [PRB: WM_QUERYOPEN Incorrectly Prototyped in WINDOWSX.H](#)
-  [INF: Using SetThreadLocale\(\) for Language Resources](#)
-  [INF: Compile Errors Caused by Missing Option -D_X86](#)
-  [PRB: Running Early Apps Results in Error w/ RtlExAllocateHeap](#)
-  [INF: Undocumented Warning C4509](#)
-  [INF: Example of Importing Functions](#)
-  [PRB: Error in Win32 SDK Install Program MANUAL.BAT](#)
-  [PRB: Destructor for Class in a DLL Called Twice](#)
-  [INF: Choosing the Debugger That the System Will Spawn](#)
-  [INF: Symbolic Information for System DLLs](#)
-  [INF: Cannot Load <exe> Because NTVDM Is Already Running](#)
-  [PRB: Win32 SDK and VC++ NT Help Files Are Incompatible](#)
-  [INF: Development Tools Do Not Accept Unicode Text](#)
-  [INF: Viewing Globals Out of Context in WinDbg](#)
-  [PRB: RC Does Not Support DATE or TIME](#)
-  [PRB: Unable to Freeze One Thread in WinDbg](#)
-  [PRB: WinDbg FIND Dialog Box Slows Down the System](#)
-  [INF: Debugging the Win32 Subsystem](#)
-  [INF: Differences Between the Win32 SDK and 32-Bit VC++](#)
-  [INF: Listing the Named Shared Objects](#)
-  [INF: Additional Remote Debugging Requirement](#)
-  [PRB: Problems with the Microsoft Setup Toolkit](#)



Win32s Topics



Sample Program Descriptions

 [Win32 SDK Knowledge Base](#)



Base Topics



GDI Topics



Networking Topics






















User Topics



Tools Topics



Win32s Topics

-  [INF: Win32s Stacks Limited to 128K](#)
 -  [INF: Description of Win32s API](#)
 -  [INF: Use 16-Bit .FON Files for Cross-Platform Compatibility](#)
 -  [INF: Support for Sleep\(\) on Win32s](#)
-  [INF: Win32s Translated Pointers Guaranteed for 32K](#)
 -  [INF: Calling a Win32 DLL from a Windows 3.1 Application](#)
 -  [INF: Win32s Message Queue Checking](#)
 -  [PRB: _getcwd\(\) Returns Incorrect Information Under Win32s](#)
 -  [PRB: GetVersion\(\) Returns Invalid Value Under Win32s](#)
 -  [INF: Debugging Win32s Applications](#)
 -  [INF: GetCommandLine\(\) Under Win32s](#)
 -  [INF: Win32s Cannot Support _environ in DLLs](#)
 -  [INF: Debugging Universal Thunks](#)
-  [INF: Using Windows Sockets Under Win32s and WOW](#)
 -  [INF: Win32s and Windows NT Timer Differences](#)
 -  [INF: Using Serial Communications Under Win32s](#)
 -  [INF: Using VxDs and Software Interrupts Under Win32s](#)
 -  [INF: Getting Resources from 16-Bit DLLs Under Win32s](#)
 -  [INF: Sharing Memory Between 32-Bit and 16-Bit Code on Win32s](#)



Sample Program Descriptions

 [Win32 SDK Knowledge Base](#)



Base Topics



GDI Topics



Networking Topics



User Topics



Tools Topics



Win32s Topics



Sample Program Descriptions

-  [Sample: Security API Functions Demonstration](#)
 -  [Sample: Saving/Loading Bitmaps in .DIB Format on MIPS](#)
 -  [Sample: Common Dialog DLL](#)
 -  [Sample: Determining Drive and File System Type](#)
-  [Sample: Walking a Directory Tree](#)
 -  [Sample: World Coordinate Transform](#)
 -  [Sample: AngleArc Demonstration Program](#)
 -  [Sample: Using GetDIBits\(\) for Retrieving Bitmap Information](#)
 -  [Sample: Writing NTSD Extensions](#)
 -  [SAMPLE: Process API Functions Example](#)
 -  [Sample: Using Thread API Functions](#)
 -  [Sample: Demonstration of Using System Info API](#)
 -  [Sample: StretchBlt Demonstration](#)
 -  [Sample: Creating Resource-Only DLLs](#)
 -  [SAMPLE: Standard DLL & Ex. of Creating a Custom Control Class](#)
 -  [Sample: Using Region-Related API Functions](#)
-  [Sample: PlgBlt Demonstration](#)
 -  [SAMPLE: Using Graphic Paths Demonstration](#)
 -  [SAMPLE: PolyBezier\(\) Demonstration](#)
 -  [Sample: Demonstration of Setting File Attributes](#)
 -  [Sample: GetDeviceCaps\(\) Demonstration Program](#)
 -  [Sample: PolyDraw Function Demonstration](#)
 -  [SAMPLE: Simple DLL Demonstration](#)
 -  [Sample: Using Timers in Windows NT](#)
 -  [Sample: Using Anonymous Pipes to Capture Child Process Output](#)
 -  [Sample: Virtual Memory API Function Demonstration](#)
 -  [Sample: SUBCLASS Program Demonstration](#)
 -  [Sample: CreateProcess\(\) Priority Demonstration](#)
-  [Sample: Code Demonstration to Put a DACL on Floppy Disk Drives](#)
 -  [Sample: MaskBlt Function Demonstration](#)
 -  [Sample: WNet API Function Demonstration](#)
 -  [Sample: Demonstration of Setting Console Text Color](#)

-  [Sample: Asynchronous I/O Demonstration](#)
-  [Sample: Communications API Function Demonstration](#)
-  [Sample: Demonstration of the Console API Functions](#)
-  [Sample: Distributed Bounded Buffer Solution \(DBBS\)](#)
-  [Sample: Demonstrating GDI and User APIs in Fractals](#)
-  [SAMPLE: Demonstration of the Win32 Font API Functions](#)
-  [SAMPLE: A Simple Service](#)
-  [SAMPLE: Enhanced Metafile Editor](#)
-  [SAMPLE: File I/O API Functions Demonstration](#)
-  [Sample: Demonstration of Journal Hooks Under Win32](#)
-  [Sample: Dynamic Dialog Box Creation](#)
-  [SAMPLE: Win32s Universal Thunks](#)
-  [Sample: How to Reboot or Shut Down Programmatically](#)
-  [Sample: Creating a WinDbg Extension](#)
-  [Sample: DDEML API Demonstration](#)
-  [Sample: Demonstration of the Win32 Debug API](#)
-  [Sample: GUIGREP File Manager Extension Sample](#)
-  [Sample: Demonstration of Using GetLocaleInfoW](#)
-  [Sample: Multiple Document Interface Demonstration](#)
-  [Sample: How to Share Memory Between Processes](#)
-  [Sample: Demonstrating the Creation of Multiple Threads](#)
-  [Sample: Constructing and Using a Message Table Resource](#)
-  [Sample: Sharing Named Memory Between Two Processes](#)
-  [Sample: Platform Detection](#)
-  [Sample: Demonstration of Printing with Windows NT](#)
-  [Sample: Named Pipe Client/Server Demonstration](#)
-  [Sample: Read/Write Synchronization Demonstration](#)
-  [Sample: Using API Functions to Access the Registry](#)
-  [Sample: Fractal Screen Saver Demonstration](#)
-  [Sample: Using Semaphores to Control Threads](#)
-  [Sample: Demonstration of Opening and Terminating a Process](#)
-  [Sample: Using TLS to Store Thread-Specific Data in a DLL](#)
-  [Sample: World Coordinate Transforms](#)
-  [SAMPLE: Primitive Drag and Drop Unicode Input Method](#)
-  [Sample: WINDIFF Source Included as an SDK Sample](#)
-  [SAMPLE: Monitoring System Events](#)
-  [SAMPLE: Examining Security Descriptors \(SDs\)](#)
-  [Sample: Using Based Pointers to Share Memory](#)
-  [SAMPLE: Event Logging](#)
-  [Sample: Combo Boxes and Owner-Draw Techniques](#)
-  [SAMPLE: Demonstration of the Windows Sockets API](#)

Article ID: Q94088

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

I have set a WSAAsyncSelect() call to notify me of read (FD_READ) and disconnection (FD_CLOSE). When a read call is posted on my message queue, I continually read from the socket until there are no more characters waiting. After each read, I use a select() call to determine if more data needs to be read. However, after a while, the notifications stop coming. Why is this?

CAUSE

The message queue must be cleared of extraneous notification messages for each read notification message.

RESOLUTION

Call WSAAsyncSelect(sockt, hWnd, 0, 0) to clear the message queue for each read notification.

More Information:

Sample Code

WSA_READCLOSE:

```
if (WSAGETSELECTEVENT( lParam ) == FD_READ) {

    FD_ZERO( &readfds );
    FD_SET( sockt, &readfds);

    timeout.tv_sec = 0;
    timeout.tv_usec = 0;

    /* Clear the queue of any extraneous notification messages. */

    WSAAsyncSelect( sockt, hWnd, 0, 0);

    while (select(0, &readfds, NULL, NULL, &timeout) != 0) {
        recv(sockt, &ch, 1, 0);
    }

    /* Reset the message notification. */

    WSAAsyncSelect( sockt, hWnd, WSA_READCLOSE, FD_READ | FD_CLOSE);
}
```


Additional reference words: 3.10 3.1

INF: Two Types of Priority Control Added

Article ID: Q91129

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Due to overwhelming demand, Microsoft added two new types of priority control to the existing types:

```
THREAD_PRIORITY_TIME_CRITICAL
THREAD_PRIORITY_IDLE
```

In addition, a new priority class has been added: `REALTIME_PRIORITY_CLASS`. This affects `CreateProcess()`, `SetPriorityClass()`, and `GetPriorityClass()`. The `SeIncreaseBasePriorityPrivilege` is required to use this class:

Therefore, the priority spectrum is organized as follows.

Fixed Priorities

```
31 - Realtime Class, THREAD_PRIORITY_TIME_CRITICAL
    .
    .
26 - Realtime Class, THREAD_PRIORITY_HIGHEST
25 - Realtime Class, THREAD_PRIORITY_ABOVE_NORMAL
24 - Realtime Class, THREAD_PRIORITY_NORMAL
23 - Realtime Class, THREAD_PRIORITY_BELOW_NORMAL
22 - Realtime Class, THREAD_PRIORITY_LOWEST
    .
    .
16 - Realtime Class, THREAD_PRIORITY_IDLE
```

Variable Priorities

```
15 - Any Class, THREAD_PRIORITY_TIME_CRITICAL,
    High Class, THREAD_PRIORITY_HIGHEST
14 - High Class, THREAD_PRIORITY_ABOVE_NORMAL
13 - High Class, THREAD_PRIORITY_NORMAL
12 - High Class, THREAD_PRIORITY_BELOW_NORMAL
11 - High Class, THREAD_PRIORITY_LOWEST
    Foreground Normal Class, THREAD_PRIORITY_HIGHEST
10 - Foreground Normal Class, THREAD_PRIORITY_ABOVE_NORMAL
 9 - Foreground Normal Class, THREAD_PRIORITY_NORMAL
    Background Normal Class, THREAD_PRIORITY_HIGHEST
 8 - Foreground Normal Class, THREAD_PRIORITY_BELOW_NORMAL
    Background Normal Class, THREAD_PRIORITY_ABOVE_NORMAL
 7 - Foreground Normal Class, THREAD_PRIORITY_LOWEST
```

- 6 - Background Normal Class, THREAD_PRIORITY_NORMAL
- Idle Class, THREAD_PRIORITY_HIGHEST
- 5 - Background Normal Class, THREAD_PRIORITY_LOWEST
- Idle Class, THREAD_PRIORITY_ABOVE_NORMAL
- 4 - Idle Class, THREAD_PRIORITY_NORMAL
- 3 - Idle Class, THREAD_PRIORITY_BELOW_NORMAL
- 2 - Idle Class, THREAD_PRIORITY_LOWEST
- 1 - Any Class, THREAD_PRIORITY_IDLE

More Information:

Within a class, the thread priority controls allow you to make two points around the normal priority for the class:

- 24 - Realtime Base
- 13 - High Base
- 9 - Forground Normal Base
- 7 - Background Normal Base
- 4 - Idle Base

In addition, two new thread controls allow you to position a thread as far from your base as possible in either the dynamic or fixed priority ranges:

- 31 - Time-critical fixed priority
- 16 - Idle fixed priority
- 15 - Time-critical variable priority
- 1 - Idle variable priority

Note that using priorities above 11 will interfere with the normal operation of the system. Using any of the real-time priorities may cause disk caches to not flush, hang the mouse, and so forth. Extreme care should be exercised whenever raising priority.

Additional reference words: 3.10 3.1

INF: FILE_FLAG_WRITE_THROUGH and FILE_FLAG_NO_BUFFERING
Article ID: Q99794

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.10
-

Summary:

The FILE_FLAG_WRITE_THROUGH flag for CreateFile() causes any writes made to that handle to be written directly to the file without being buffered. The data is cached (stored in the disk cache); however, it is still written directly to the file. This method allows a read operation on that data to satisfy the read request from cached data (if it's still there), rather than having to do a file read to get the data. The write call doesn't return until the data is written to the file. This applies to remote writes as well--the network redirector passes the FILE_FLAG_WRITE_THROUGH flag to the server so that the server knows not to satisfy the write request until the data is written to the file.

The FILE_FLAG_NO_BUFFERING takes this concept one step further and eliminates all read-ahead file buffering and disk caching as well, so that all reads are guaranteed to come from the file and not from any system buffer or disk cache. When using FILE_FLAG_NO_BUFFERING, disk reads and writes must be done on sector boundaries, and buffer addresses must be aligned on disk sector boundaries in memory.

These restrictions are necessary because the buffer that you pass to the read or write API is used directly for I/O at the device level; at that level, your buffer addresses and sector sizes must satisfy any processor and media alignment restrictions of the hardware you are running on.

More Information:

If you have a situation where you want to flush all open files on the current logical drive, this can be done by:

```
hFile = CreateFile("\\\\.\\c:", ....);  
FlushFileBuffers(hFile);
```

This method causes all buffered write data for all open files on the C: partition to be flushed and written to the disk. Note that any buffering done by anything other than the system is not affected by this flush; any possible file buffering that the C run time is doing on files opened with C run-time calls is unaffected.

Additional reference words: 3.10

Sample: Security API Functions Demonstration

Article ID: Q85397

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SIDCLN sample demonstrates some of the Win32 security API functions, and provides a sample of how a utility could be written that recovers on-disk resources remaining allocated to deleted user accounts.

More Information:

The on-disk resources recovered are:

Files that are still owned by accounts that have been deleted are assigned ownership to the account logged on when this sample is run.

ACEs for deleted accounts are edited (deleted) out of the ACLs of files to which the deleted accounts had been granted authorizations (eg., Read access)

It may be that running this sample as a utility has no practical value in many environments, as the number of files belonging to deleted user accounts will often be quite small, and the number of bytes recovered on disk by editing out ACEs for deleted accounts may well not be worth the time it takes to run this sample. The time it takes to run this sample may be quite significant when processing an entire hard disk or partition

Note: This sample is not a supported utility.

TO RUN:

You must log on using an account, such as Administrator, that has the privileges to take file ownership and edit ACLs

The ACL editing part of this sample can only be exercised for files on a partition that has ACLs NT processes: NTFS

Typical test scenario: Create a user account or two, log on as each of these accounts in turn, while logged on for each account, go to an NTFS partition, create a couple of files so the test accounts each own a few files, use the file manager to edit permissions for those files so that each

test user has some authorities (e.g., Read) explicitly granted for those files. Logon as Administrator, authorize each test user to a few Administrator-owned files. Delete the test accounts. Run the sample in the directories where you put the files the test accounts owned or were authorized to.

PRB: SetConsoleOutputCP() Not Functional
Article ID: Q99795

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

SYMPTOMS

SetConsoleOutputCP() apparently has no effect.

CAUSE

SetConsoleOutputCP() was designed to change the mapping of the 256 8-bit character values into the glyph set of a fixed-pitch Unicode font, rather than loading a separate, non-Unicode font for each call to SetConsoleOutputCP(). Unfortunately, a fixed-pitch Unicode font was not available by release time, so you can't view the effects of the SetConsoleOutputCP() application programming interface (API) because the currently available console fonts are not Unicode fonts.

STATUS

Microsoft has confirmed this to be a limitation in Windows NT version 3.1. We will post new information here in the Microsoft Knowledge Base as it becomes available.

Additional reference words: 3.10

Sample: Common Dialog DLL

Article ID: Q81703

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A sample demonstrating the use of all of the common dialog box functions in the Win32 API is now available.

More Information:

Each dialog box is demonstrated being used in three different ways: standard, using a hook function, and using a modified template.

Additional reference words:

ChooseColor, ChooseFont, GetOpenFileName, GetSaveFileName

Sample: Determining Drive and File System Type

Article ID: Q81719

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A demonstration of how the `GetDriveType` and `GetVolumeInformation` functions determine all logical drives on a system, their disk type (local, remote, CD-ROM, and so on), and their file system type (FAT, HPFS, and so on) is available in a sample file named `DRIVES`.

More Information:

The additional API functions, `GetLogicalDrives` and `GetLogicalDriveStrings`, are not required to determine the drive and file system type, but are included as an example of how these API functions can enhance the efficiency of disk querying API calls.

When a drive type is removable (for example, a floppy disk drive), then additional precautions are taken before accessing this drive. A validation check is made to see if media exists in the drive before proceeding. A simple test of opening any file in the root directory of the removable media drive using the `OpenFile` API function determines the media's presence. If the `OpenFile` call returns a handle, then media is present and further disk querying calls are safely made on the logical drive. If the `OpenFile` call fails, then no media is present and no further attempts to query this drive are allowed. Note: in order to eliminate an unwanted pop-up, prompting the user to insert a disk in the drive, from being generated by the operating system, the error mode is temporarily adjusted, using `SetErrorMode`, to allow any `OpenFile` errors to immediately return to the calling routine.

Sample: Walking a Directory Tree

Article ID: Q81720

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A demonstration of how to recursively find all subdirectories under the current working directory is available in a sample file named WALK.

The WALK sample can be found in the \Q_A\SAMPLES\WALK directory.

More Information:

Starting with the current working directory, a call is made to the Walk function which will find all subdirectories in the current working directory. When a subdirectory is found, the current working directory is changed to this subdirectory and another, recursive call is made to Walk, which again will find all subdirectories in this new current working directory. Once all subdirectories for the current working directory have been found, the current working directory is changed up one level (..). When the original current working directory is re-entered, then the recursive process stops.

Additional reference words:

FindFirstFile, GetCurrentDirectory, SetCurrentDirectory
FindNextFile, GetFileAttributes

PRB: ERROR_SEM_TIMEOUT Not Documented

Article ID: Q98720

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

A call to WaitNamedPipe() fails, and GetLastError() returns:

ERROR_SEM_TIMEOUT (121)

CAUSE

The call to WaitNamedPipe() fails due to an elapsed time-out interval.

STATUS

This probable error is missing from the documentation. This probable error return is also missing for all functions that might time out, such as WaitForSingleObject().

Additional reference words: 3.10

SAMPLE: Process API Functions Example

Article ID: Q81825

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The PROCESS sample application provides a simple interface to the `CreateProcess()` and `TerminateProcess()` functions. To create a process, the user is presented with a common dialog box for selecting a file. In this case, the file must have an extension of `.EXE`. Processes that are started are presented in a list box. Any of the processes can be selected in the list box and then terminated.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at (800) 227-4679, ext 11771.

Warning: "TerminateProcess() is used to cause all of the threads within a process to terminate. While `TerminateProcess()` will cause all threads within a process to terminate, and will cause an application to exit, it does not notify DLLs that the process is attached to that the process is terminating. `TerminateProcess()` is used to unconditionally cause a process to exit. It should only be used in extreme circumstances. The state of global data maintained by DLLs may be compromised if `TerminateProcess()` is used rather than `ExitProcess()`."

Additional reference words: 3.10

Sample: Creating Resource-Only DLLs

Article ID: Q85915

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The RESDLL sample shows how to create a resource-only dynamic link library (DLL). In short, this is accomplished by creating and resource-compiling a resource (.RC) file, and then linking it correctly.

The MAIN.EXE program tests THE_DLL.DLL by loading it and referencing the DLL's icon, cursor, and bitmap. The icon and cursor are used by the registered window class, and the bitmap is used in painting the client area.

INF: Objects Inherited Through a CreateProcess Call
Article ID: Q83298

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following is a list of objects that are inherited through the CreateProcess() call.

More Information:

The objects inherited by a process are those objects that you can get a handle to, and that you can use the CloseHandle() function on. These objects include the following:

- Processes
- Events
- Semaphores
- Mutexes
- Files (including file mappings)
- Console input or output

However, the new process will only inherit objects that were marked inheritable by the old process.

These are duplicate handles. Each process maintains memory for its own handle table. If one of the processes modifies its handle (for example, closes it or changes the mode for the console handle), other processes will not be affected.

Processes will also inherit environment variables, the current directory, and priority class.

Sample: Demonstration of Setting File Attributes
Article ID: Q85917

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SETINFO sample shows how to get and set information on file date, time, attributes, and size. SETINFO demonstrates much of the functionality of OS/2's DosQFileInfo() and DosSetFileInfo().

More specifically, the SETINFO sample shows how to set file attributes and how to modify file and date times (and how to do the conversions from file time to DosTime, and so on). To use the sample file, enter a filename in the appropriate edit field and choose the Get Info. button. To set file attributes or file date and time information, modify the values in the edit fields and check buttons, and choose the Set Info. and Set Attr. buttons. To slow down the return code reporting, enter a larger value into the time edit field, and choose the Set Time button.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

INF: Correct Use of Try/Finally

Article ID: Q83670

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Try/finally, used correctly, helps to provide a robust application. However, if used incorrectly it can cause unnecessary overhead. any flow of control out of the try body of try/finally is an abnormal termination that can cause hundreds of instructions to be executed on an x86 system, and thousands on a MIPS machine, even if control leaves the try body via a control statement on the very last statement of the try body. The language definition states that control must leave the try body sequentially for normal termination to occur (that is, execution falls through the bottom of the try body).

The following sample demonstrates an incorrect use of try/finally:

```
/* Incorrect use of try/finally */
```

```
VOID
function (
    DWORD ... P1,
    .
    DWORD ... Pn
)
{
    try {
        if (...) {
            .
            .
            return;
        }
        .
        .
    } finally {
        .
        .
    }
    return;
}
```

The overhead can be avoided in the above example by moving the return AFTER the end of the finally clause. The following provides more detail on the correct use of try/finally.

More Information:

Execution of a termination handler due to abnormal termination of a try body is expensive. Abnormal termination occurs when control leaves a try body in any way other than by falling through the bottom. Intentionally branching out of a try body is still an abnormal termination.

In the above example, abnormal termination of the try body occurs if the return in the middle of the try body is executed. If the predicate of the if is false, then extremely efficient execution of the finally clause occurs because this is not abnormal termination and the finally clause is called directly by inline code.

When abnormal termination occurs hundreds to thousands of instructions are executed because an unwind must be executed, which must search backward through frames to determine if any termination handlers should be called. On an x86 system, this executes the C run-time handler and examines the handler list. On a MIPS machine, this also causes the function table to be searched and the prologue of each intervening function to be executed backwards interpretively.

You should always avoid the execution of a termination handler as a result of the abnormal termination of a try body by a return, or other direct flow of control out of the try body. Abnormal termination occurs whenever control leaves the try body other than by falling through the bottom. This can occur because of a return, goto, continue, or break. It can also occur because of an exception, which presumably cannot be avoided.

In the above example, abnormal termination in the nonexception case can be eliminated easily as follows:

```
/* Correct use of try/finally */
```

```
VOID
function (
    DWORD ... P1,
    .
    .
    DWORD ... Pn
)
{
    try {
        if (...) {
            .
            .
        } else {
            .
            .
        }
    } finally {
        .
        .
    }
    return;
}
```

```
}
```

Now both clauses of the if fall through to the termination handler in all but exceptional cases and execute the termination handler in the most efficient way. This also has the same logical execution as the previous sample.

In summary, the correct use of try/finally is a powerful method to help you write robust applications. Care should be taken to ensure the correct use of try/finally.

SAMPLE: Simple DLL Demonstration

Article ID: Q83932

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SIMPLDLL sample provides a generic DLL template. Also included are two small test applications, LINKTEST and LOADTEST, which demonstrate load-time linking (to a DLL import library) and dynamic loading, respectively.

More Information:

THE_DLL contains a skeleton DLL (dynamic-linked library) entry point and five exported functions with varying parameter lists. A resource file (containing a dialog box template) is also used.

LINKTEST is a small application that links with the THE_DLL's import library, and allows the user to make calls into THE_DLL (via menu item selections).

LOADTEST is a small application that loads THE_DLL at run time and calls the GetProcAddress function to retrieve the addresses of THE_DLL's exported functions. Again, the user is allowed to make calls into THE_DLL.

Additional reference words: GetModuleFileName, LoadLibrary, GetProcAddress

Sample: Virtual Memory API Function Demonstration

Article ID: Q85919

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

VIRTMEM is a sample of the various virtual memory API functions available under Win32.

When you start the application, you are initially given a RESERVED page of virtual memory. You can change the protection and state of the page through menu selections. Check marks will appear in the menu items to indicate the current state and protection on the page. More in-depth information regarding the page can be obtained by selecting the Show Page menu item.

The Lock menu item allows you to lock and unlock the page in memory.

The application also uses structured exception handling and allows you to try and write to the page in its various states and protections. To do this, select the Test menu option.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

INF: NT Consoles Do Not Support ANSI Escape Sequences
Article ID: Q84240

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Windows NT currently does not support ANSI escape sequences. This should affect very few uses (for example, changing the color of the prompt) and a very limited number TTY-type programs that rely on the console for escape support to be provided.

This feature is under review and is being considered for future releases.

INF: Setting the Console Configuration

Article ID: Q105674

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

To create a command prompt with custom features such as

```
Settings
Fonts
Screen Size and Position
Screen Colors
```

create a new entry in the Program Manager for CMD.EXE (suppose that the description is CUSTOM), choose these items from the CMD system menu, and select Save Configuration in each dialog box. The settings are saved in the registry under

```
HKEY_CURRENT_USER\
    Console\
        custom
```

and are used when starting the CUSTOM command prompt from the Program Manager or when specifying:

```
start "custom"
```

This behavior is really a convenient side effect of

```
start <string>
```

which sets the title in the window title bar. When you create a new console window with the START command, the system looks in the registry and tries to match the title with one of the configurations stored there. If it cannot find it, it defaults to the values stored in:

```
HKEY_CURRENT_USER\
    Console\
        Configuration
```

This functionality can be duplicated in your own applications using the registry application programming interface (API).

For more information, please see the "Registry and Initialization Files" overview and the REGISTRY sample.

Additional reference words: 3.10

Sample: CreateProcess() Priority Demonstration

Article ID: Q84539

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The sample STARTP demonstrates how to start a new process at a given default priority. It is a functional replacement for the "start" command, but with added features.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

STARTP starts a separate window to run a specified program. The following is the command syntax for STARTP.

```
STARTP [/Ttitle] [/Dpath] [/h] [/l] [/min] [/max]
        [/c] [program] [parameters]
```

title	Title to display in window title bar. Put entire parameter in quotation marks to include spaces in the title; for example, startp "/Ttest job".
path	Starting directory.
h	Set default to high priority.
l	Set default to low priority.
min	Start window minimized.
max	Start window maximized.
c	Use current console instead of creating a new console.
program	A program to run as either a GUI application or a console application.
parameters	These are the parameters passed to the program.

Note that the priority parameters may have no effect if the program changes its own priority.

Sample: Demonstration of Setting Console Text Color
Article ID: Q87329

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The CONSOLEC sample illustrates the use of the SetConsoleTextAttribute() and GetConsoleScreenBufferInfo() functions to set the console text color attributes.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

This sample also functions as a utility to set the text color of the console window. The command syntax for the utility is as follows:

```
COLOR FOREGROUND [BACKGROUND]
```

FOREGROUND and BACKGROUND are the new text color selections for the current console. If the utility is invoked without any options, the utilities syntax and a table of the possible color choices is displayed. The BACKGROUND selection is optional, and thus just the FOREGROUND text color can be changed.

Possible colors are: black, blue, green, cyan, red, magenta, yellow and white. Each of these can be selected as the FOREGROUND or the BACKGROUND color. Selection of the same color for both the FOREGROUND and the BACKGROUND is not permitted. The color options are not case sensitive, and only the first unique characters are necessary to select the color. For example

```
COLOR BLU W
```

will select blue on white text color attributes.

The text color attribute changes only affect new console output. Thus, text in the console buffer before the utility is invoked retains its original color attributes.

Additional reference words:

Sample: Asynchronous I/O Demonstration

Article ID: Q87330

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The EVENT sample demonstrates performing asynchronous I/O in Win32. In Win32 you can do this in two ways. One way is similar to OS/2 where a thread is spawned that performs the I/O and returns. With Win32, when you create a file, it signals to the system that you want to perform I/O asynchronously. Then, when ReadFile(), for example, is going to take a significant amount of time to complete, an ERROR_IO_PENDING error is generated, signaling you to do other tasks until you NEED the data, at which time you can use the GetOverlappedResults() function, which will finish the I/O.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

Note that this activity has taken place without the need of an additional thread to perform the work. This sample touches only on the capabilities of what one can do with the new overlapped I/O functions. For example, an application that uses pipes to communicate over the network to other clients can create these file handles with the overlapped flag. Then, instead of blocking and waiting for a connection, the server application can go about doing useful tasks waiting for the pipe to enter the "signaled" state. In addition, you can perform more than one operation on this handle at one time, such as reading and writing to the same file.

All this power does not come without some responsibilities on the programmer's side. First, the system does not keep track of the system file pointers. In addition, you cannot use the data until the system responds by setting an event to a signaled state.

In the first case this just means you need to keep track of the value "lpNumberOfBytesTransferred" returned by GetOverlappedResult() and update the OVERLAPPED structure with this information. This OVERLAPPED structure will then be passed into the Read/WriteFile() function, which will use this as the offset to the starting point for the I/O operation. The first call to Read/WriteFile() will normally then have the offset fields in the OVERLAPPED structure set to zero.

The second case should be used as a criteria of whether to use this type of I/O. If you need the data before you can do anything else, use

normal synchronous I/O and let the system handle the details for you. This also demonstrates an important reason for using an EVENT to wait on rather than the file handle. While both are allowed in a multithread application, one cannot guarantee that the thread that set the handle to the signaled state will be the one returning from the GetOverlappedResult() because each thread is using the same handle to wait on.

To keep this sample focused, the user interface is simple. To run this sample at the command prompt, type:

```
ASYNC_IO <In_file> <Out_file>
```

In_file and Out_file are place holders. As this is implemented, you cannot write over an existing file. While this is up and running, you will see vital statistics such as, such as the following:

- When I/O is pending
- How many bytes are transferred
- End of file

Sample: Demonstration of the Console API Functions

Article ID: Q87332

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The CONSOLE sample demonstrates the Win32 console API functions. The program takes no parameters; to start, just run CONSOLE.EXE. After starting the sample, you can click on one of the functions on the screen to get a demonstration of that function. When viewing a demo of the function, the title of the console window is changed to show the name of the source file where that demo function resides. This should make it easy to find the sample code where the function of interest resides.

Please note that some of the demos cover multiple APIs, so some of the menu choices run the same demo.

INF: Replacing the Shell (Program Manager)

Article ID: Q100328

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

To replace the current shell, change the following registry key:

```
HKEY_LOCAL_MACHINE\  
  SOFTWARE\  
    MICROSOFT\  
      Windows NT\  
        Current Version\  
          Winlogon\  
            Shell
```

Note that Program Manager combines the functionality of Program Manager and Task Manager (the Task Manager installed is not actually run). Therefore, if the new shell does not replace the Task Manager functionality, the replacement string should contain both the new shell name and TASKMAN.EXE, separated by commas.

To update the string that is retrieved when calling `GetPrivateProfileString()`, change the string in the following registry key:

```
HKEY_LOCAL_MACHINE\  
  SOFTWARE\  
    MICROSOFT\  
      Windows NT\  
        Current Version\  
          WOW\  
            Boot\  
              Shell
```

The duplicate entry is for compatibility with Windows 3.1.

More Information:

`WritePrivateProfileString()` changes the following registry key:

```
HKEY_LOCAL_MACHINE\  
  SOFTWARE\  
    MICROSOFT\  
      Windows NT\  
        Current Version\  
          WOW\  
            Boot\  
              Shell
```

It does not have the desired effect of actually changing the Shell.

Additional reference words: 3.10

INF: Four New Win32 Memory-Handling APIs
Article ID: Q94146

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

We have added four new Win32 APIs and changed the definitions of a few NT rtl APIs. The changes were designed so that all of our code could move, copy, fill, and zero memory using the fastest mechanism on each platform.

Note: From now on, please use the following APIs:

NT Code Win 32 Applications

Copy Memory - RtlCopyMemory() CopyMemory()
Moving Memory - RtlMoveMemory() MoveMemory()
Zeroing Memory - RtlZeroMemory() ZeroMemory()
Filling Memory - RtlFillMemory() FillMemory()

The difference between RtlCopyMemory() and RtlMoveMemory() is that RtlCopyMemory() does not handle overlapped copies; it is equivalent to memcpy(). RtlMoveMemory() handles overlapped copies (similar to memmove()).

If you look closely at the changes made to support the memory movers, you will see that on the x86 platform, the movers are defined to be identical to the C run-time intrinsic movers. The x86 compilers will inline these calls for us. On MIPS, these calls are vectored (through forwarders in KERNEL32.DLL) to code in NTDLL.DLL.

INF: Secure Erasure Under Windows NT

Article ID: Q94239

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

File systems under Windows NT currently have virtual secure erasure (when a file is deleted, the data is no longer accessible through the operating system). Although the bits could still be on disk, NT will not allow access to them.

NTFS does this by keeping a high-water mark, for each file, of bytes written to the file. Everything below the line is real data, anything above the line is (on disk) random garbage that used to be free space, but any attempt to read past this high-water mark returns all zeros.

Other reusable objects are also protected. For example, all the memory pages in a process's address space are zeroed (unlike the file system, a process may directly access its pages, and thus the pages must be actually zeroed rather than virtually zeroed).

Note that file system security assumes physical security; in other words, if a person has physical access to a machine and can boot an alternative operating system and/or add custom device drivers and programs, he/she can always get direct access to the bits on disk.

Additional reference words: 3.10 3.1

INF: First and Second Chance Exception Handling

Article ID: Q105675

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Structured exception handling (SEH) takes a little getting used to, particularly when debugging. It is common practice to use SEH as a signaling mechanism. Some application programming interfaces (APIs) register an exception handler in anticipation of a failure condition that is expected to occur in a lower layer. When the exception occurs, the handler may correct or ignore the condition rather than allowing a failure to propagate up through intervening layers. This is very handy in complex environments such as networks where partial failures are expected and it is not desirable to fail an entire operation simply because one of several optional parts failed. In this case, the exception can be handled so that the application is not aware that an exception has occurred.

However, if the application is being debugged, it is important to realize that the debugger will see all exceptions before the program does. This is the distinction between the first and second chance exception. The debugger gets the "first chance," hence the name. If the debugger continues the exception unhandled, the program will see the exception as usual. If the program does not handle the exception, the debugger will see it again (the "second chance"). In this latter case, the program normally would have crashed had the debugger not been present.

If you do not want to see the first chance exception in the debugger, then disable the feature. Otherwise, during execution, when the debugger gets the first chance, continue the exception unhandled and allow the program to handle the exception as usual. Check the documentation for the debugger that you are using for descriptions of the commands to be used.

Additional reference words: 3.10

INF: Thread Local Storage Overview

Article ID: Q94804

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Thread local storage (TLS) is a method by which each thread in a given process is given a location(s) in which to store thread-specific data.

Four functions exist for TLS: `TlsAlloc()`, `TlsGetValue()`, `TlsSetValue()`, and `TlsFree()`. These functions manipulate TLS indexes, which refer to storage areas for each thread in a process. A given index is valid only within the process that allocated it.

A call to `TlsAlloc()` returns a global TLS index. This one TLS index is valid for every thread within the process that allocated it, and should therefore be saved in a global or static variable.

Thread local storage works as follows: When `TlsAlloc()` is called, every thread within the process has its own private DWORD-sized space reserved for it (in its stack space, but this is implementation-specific). However, only one TLS index is returned. This single TLS index may be used by each and every thread in the process to refer to its unique space that `TlsAlloc()` reserved for it.

For this reason, `TlsAlloc()` is often called only once. This is convenient for DLLs, which can distinguish between `DLL_PROCESS_ATTACH` (where the first process's thread is connecting to the DLL) and `DLL_THREAD_ATTACH` (subsequent threads of that process are attaching). For instance, the first thread calls `TlsAlloc()` and stores the TLS index in a global or static variable, and every other thread that attaches to the DLL refers to the global variable to access their local storage space.

Although one TLS index is usually sufficient, a process may have up to `TLS_MINIMUM_AVAILABLE` indexes (guaranteed to be greater than or equal to 64).

Once a TLS index has been allocated (and stored), the threads within the process may use it to set and retrieve values in their storage spaces. A thread may store any DWORD-sized value in its local storage (for example, a DWORD value, a pointer to some dynamically allocated memory, and so forth). The `TlsSetValue()` and `TlsGetValue()` functions are used for this purpose.

Note that the `TlsSetValue()` and `TlsGetValue()` functions are optimized for speed, and therefore do minimal parameter checking. Therefore, it is up to the programmer to ensure that the index passed to these functions is valid.

A process should free TLS indexes with `TlsFree()` when it has finished using them. However, if any threads in the process have stored a pointer to dynamically allocated memory within their local storage spaces, it is important to free the memory or retrieve the pointer to it before freeing the TLS index, or it will be lost.

Using thread local storage may be a little more convenient than other solutions, and will be more profitable in the future. In later releases of NT, the compiler will include a keyword and compiler switches to make thread local storage operations more automatic, rather than through an application programming interface (API) layer.

For more information, see the Win32 Software Development Kit (SDK) "Programming Techniques" help file under the TLS API.

Example

Thread A within a process calls `TlsAlloc()`, and stores the index returned in the global variable `TlsIndex`:

```
TlsIndex = TlsAlloc();
```

Thread A then allocates 100 bytes of dynamic memory, and stores it in its local storage:

```
TlsSetValue( TlsIndex, malloc(100) );
```

Thread A creates thread B, which stores a handle to a window in its local storage space referred to by `TlsIndex`.

```
TlsSetValue( TlsIndex, (LPVOID)hSomeWindow );
```

Note that `TlsIndex` refers to a different location when thread B uses it, than when thread A uses it. Each thread has its own location referred to by the same value in `TlsIndex`.

Thread B may terminate safely because it does not need to specifically free the value in its local storage.

Before thread A terminates, however, it must first free the dynamically allocated memory in its local storage

```
free( TlsGetValue( TlsIndex ) );
```

and then free the TLS index:

```
if ( !TlsFree( TlsIndex ) )  
    // TlsFree() failed. Handle error.
```

Additional reference words: 3.10

INF: CPU Quota Limits Not Enforced

Article ID: Q100329

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

On page 88 of "Inside Windows NT," Table 4-1 indicates that a process object contains a quota limit for the maximum amount of processor time that the process can use.

This limit is not enforced in Windows NT version 3.1.

Additional reference words: 3.10

INF: Using GMEM_DDESHARE in Win32 Programming

Article ID: Q99114

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The `GMEM_DDESHARE` flag remains a legitimate value for `GlobalAlloc()`. This flag can be used to indicate that the memory will be used for one of the following so that the system can optimize the allocation for these special needs:

- DDE
- OLE 1.0
- Clipboard operations

However, `GlobalAlloc(GMEM_DDESHARE, ...)` cannot be used to allocate a block of memory that can be shared between processes. This flag was never intended for this purpose, even under Windows versions 3.0 and 3.1 (3.x). `GlobalAlloc(GMEM_DDESHARE, ...)` works in this case because all Windows 3.x applications share the same address space; this is not the case under Windows NT.

All allocations of global shared memory can be used within the process that they are allocated in, but another mechanism is required to share memory between processes.

Additional reference words: 3.00 3.10

INF: Direct Drive Access Under Win32

Article ID: Q100027

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

To open a physical hard drive for direct disk access (raw I/O), a Win32 device name of the form

\\.\PhysicalDriveN

is available to Win32 applications, where N is 0, 1, 2, and so forth, representing each of the physical drives in the system.

To open a logical drive, direct access is of the form

\\.\X:

where X: is a hard-drive partition letter, floppy disk drive, or CD-ROM drive.

MORE INFORMATION

=====

You can open a physical or logical drive using the CreateFile() application programming interface (API) with these device names provided that you have the appropriate access rights to the drive (that is, you must be an administrator). You must use both the CreateFile() FILE_SHARE_READ and FILE_SHARE_WRITE flags to gain access to the drive.

Once the logical or physical drive has been opened, you can then perform direct I/O to the data on the entire drive. When performing direct disk I/O, you must seek, read, and write in multiples of sector sizes of the device and on sector boundaries. Call DeviceIoControl() using IOCTL_DISK_GET_DRIVE_GEOMETRY to get the bytes per sector, number of sectors, sectors per track, and so forth, so that you can compute the size of the buffer that you will need.

Note that you cannot open a file under Win32 by using internal Windows NT object names; for instance, attempting to open a CD-ROM drive by opening

\\Device\CdRom0

does not work because this is not a valid Win32 device name. An application can use the QueryDosDevice() API to get a list of all valid Win32 device names and see the mapping between a particular

Win32 device name and an internal Windows NT object name. An application running at a sufficient privilege level can define, redefine, or delete Win32 device mappings by calling the DefineDosDevice() API.

Additional reference words: 3.10

INF: Byte-Ordering in a Data Packet Under NDIS
Article ID: Q89374

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

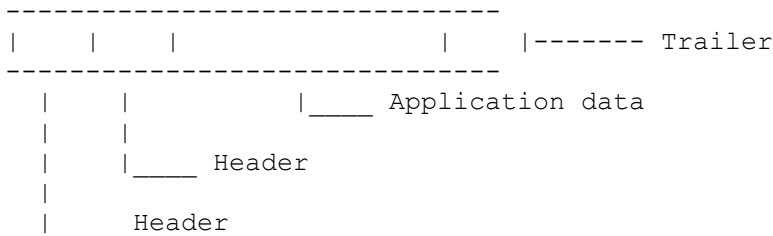
Summary:

There may be a difference in the byte order in which 16-bit words are stored in memory versus the order in which the two bytes must be transmitted onto the network as part of a data packet. This difference will depend on the processor involved and which part of the data packet the information falls under.

The information in this article is consistent with the Network Device Interface Specification (NDIS) version 3.0.

More Information:

Consider that the data will consist of header(s), the application data, and a trailer(s).



When a protocol driver or an NDIS driver sends information, it does not modify the application data, nor does it modify the ordering of bytes in integer fields.

The ordering of bytes in integer fields within the application data is the responsibility of the Remote Procedure Call (RPC) facility or another mechanism. However, the driver should be concerned with how the information in the header(s)/trailer(s) is stored. For example, the driver may be required to put a 16-bit checksum in a header. To put that integer value into the header in the format required by the network specification, the driver may need to know the type of processor that it is running on and will in any case need to follow the network standard for storing the information in the header.

If the driver needs to find out the CPU type of the machine it is running on, it can use `NdisReadConfiguration()` on the Keyword "ProcessorType" to query this. However, as of 10/25/92 this is not supported with the prerelease device driver kit (DDK), so the alternative in the meantime is `#ifdef i386, mips,` and so on.

Additional reference words: 3.10

INF: Performing a Clear Screen (CLS) in a Console Application

Article ID: Q99261

Summary:

There is no Win32 application programming interface (API) that will clear the screen in a console application. However, it is fairly easy to write a function that will programmatically clear the screen. The following function is an example:

```
void cls( HANDLE hConsole )
{
    COORD coordScreen = { 0, 0 };    /* here's where we'll home the
                                     cursor */

    BOOL bSuccess;
    DWORD cCharsWritten;
    CONSOLE_SCREEN_BUFFER_INFO csbi; /* to get buffer info */
    DWORD dwConSize;                /* number of character cells in
                                     the current buffer */

    /* get the number of character cells in the current buffer */

    bSuccess = GetConsoleScreenBufferInfo( hConsole, &csbi );
    PERR( bSuccess, "GetConsoleScreenBufferInfo" );
    dwConSize = csbi.dwSize.X * csbi.dwSize.Y;

    /* fill the entire screen with blanks */

    bSuccess = FillConsoleOutputCharacter( hConsole, (TCHAR) ' ',
        dwConSize, coordScreen, &cCharsWritten );
    PERR( bSuccess, "FillConsoleOutputCharacter" );

    /* get the current text attribute */

    bSuccess = GetConsoleScreenBufferInfo( hConsole, &csbi );
    PERR( bSuccess, "ConsoleScreenBufferInfo" );

    /* now set the buffer's attributes accordingly */

    bSuccess = FillConsoleOutputAttribute( hConsole, csbi.wAttributes,
        dwConSize, coordScreen, &cCharsWritten );
    PERR( bSuccess, "FillConsoleOutputAttribute" );

    /* put the cursor at (0, 0) */

    bSuccess = SetConsoleCursorPosition( hConsole, coordScreen );
    PERR( bSuccess, "SetConsoleCursorPosition" );
    return;
}
```

Additional reference words: 3.10 clearscreen

INF: AllocConsole() Necessary to Get Valid Handles

Article ID: Q89750

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

If a graphical user interface (GUI) application redirects a standard handle, such as stderr or stdout, and spawns a child process, the output of the child process will not be seen unless the AllocConsole() application programming interface (API) function is called before the standard handle is redirected.

More Information:

If you spawn a child process without calling AllocConsole() first, your child console window will appear on the screen and your GUI application will not be able to control this window (for example, it cannot minimize the child window). In addition, users can terminate the child process by choosing Close from the console window's Control (system) menu. This causes users to think that only the window is closed, when in actuality, the entire application is terminated. This can cause the user to lose data in the console window.

Additional reference words: 3.10

INF: How to Specify Shared and Nonshared Data in a DLL
Article ID: Q89817

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

To have both shared and nonshared data in a dynamic-link library (DLL), you need to use the new `#pragma data_seg` directive to set up a new named section. You then specify the sharing attributes for this new named data section in your `.DEF` file.

More Information:

Below is a sample of how to define a named data section in your DLL. The first line directs the compiler to include all the data declared in this section in the `.MYSEC` data segment. This means that the `iSharedVar` variable would be considered part of the `.MYSEC` data segment. By default, data will be nonshared.

Note that you must initialize all data in your named section. The `data_seg` pragma only applies to initialized data.

The third line, `"#pragma data_seg()"`, directs the compiler to reset allocation to the default data section.

```
#pragma data_seg(".MYSEC")
int iSharedVar = 0;
#pragma data_seg()
```

Below is a sample of the `.DEF` file that supports the shared and nonshared segments. This definition will set the default section `.MYSEC` to be shared. The default data section is by default non-shared, so any data not in section `.MYSEC` will be non-shared.

```
LIBRARY
SECTIONS
    .MYSEC READ WRITE SHARED
EXPORTS
    ...
```

Note: All section names must begin with a period character ('.') and must not be longer than 8 characters, including the period character.

INF: Precautions When Passing Security Attributes

Article ID: Q94839

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

All Win32 APIs that allow security to be specified take a parameter of type LPSECURITY_ATTRIBUTES as the means to attach the security descriptor. However, it is a common error to pass a PSECURITY_DESCRIPTOR type to such functions instead. Because PSECURITY_DESCRIPTOR is of type LPVOID (for opaque data-type reasons), by C language definition, it is implicitly converted to the correct type. Therefore, the compiler does not generate any warnings; however, unexpected run-time errors will result.

Below is a correct example of creating a named pipe with a security descriptor attached.

Sample Code

```
-----  
  
saSecurityAttributes.nLength = sizeof(SEcurity_ATTRIBUTES);  
saSecurityAttributes.lpSecurityDescriptor = psdAbsoluteSD;  
saSecurityAttributes.bInheritHandle = FALSE;  
  
hPipe = CreateNamedPipe(TEST_PIPE_NAME,  
                        PIPE_ACCESS_DUPLEX,  
  
(PIPE_TYPE_BYTE|PIPE_READMODE_BYTE|PIPE_WAIT),  
                        100, // maximum instances  
                        0, // output buffer, sized as needed  
                        0, // input buffer, sized as needed  
                        100, // timeout in milliseconds  
  
(LPSECURITY_ATTRIBUTES)&saSecurityAttributes);  
if (INVALID_HANDLE_VALUE == hPipe)  
{ // handle error  
}
```

Additional reference words: 3.10 3.1 lpsa psd

INF: Physical Memory Limits Number of Processes/Threads
Article ID: Q94840

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Each time Windows NT creates a process or a thread, it must allocate a certain amount of physical memory (significantly more for processes than threads) for its support. The specific amount of memory allocated has not been finalized because development is working on reducing thread/process memory requirements.

Due to the physical memory requirement of processes and threads, programs that use the CreateProcess() and CreateThread() APIs should be careful to check their return codes to detect out-of-memory conditions.

Additional reference words: 3.10

INF: Security and Screen Savers

Article ID: Q96780

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Screen savers are user-mode applications that execute in a different desktop (see the note at the end of the article). Therefore, a screen saver cannot find any other windows (if attempting to enumerate or find other windows). This design prevents unauthorized users from viewing the contents of applications displayed on the screen. For secure screen savers (those that ask for a password), this adds a further layer of protection.

Screen savers also execute in the security context of the logged-on user. A screen saver may call `ExitWindowsEx()`, to log off from or shut down the system, or any other application programming interface (API) that the logged-on user has permission to perform.

More Information:

A sample screen saver is distributed on the March Beta Software Development Kit (SDK) in the `\Q_A\SAMPLES\SCRNSAVE` subdirectory.

Note: A desktop is a virtual screen. Windows NT currently has three desktops--the main desktop, the WinLogon desktop, and a desktop for screen savers. The Win32 APIs do not allow creation of multiple desktops.

Additional reference words: 3.10

INF: Preventing the Console from Disappearing

Article ID: Q99115

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

When a console application is started from the File Manager, from the Program Manager, or by typing "start <progrname>" from the command prompt, it executes in its own console. This console disappears as soon as the application terminates, and therefore the user can't read anything written to the screen between the last pause and program exit. To resolve this problem, the programmer should pause the application before termination to allow the user to read all of the information on the screen.

It is not likely that the programmer will want to introduce this pause if the application is started directly from the command prompt, because in this situation it won't make much sense to the user. However, there is no API (application programming interface) that directly determines whether or not the application shares a console with CMD.EXE. There is a method that can be used to determine this information in most cases. When the application first starts up, call `GetConsoleScreenBufferInfo()`. If the cursor position is (0, 0), then the application has its own console, which will disappear when the application terminates. Otherwise, the application is operating within a console belonging to another program, typically CMD.EXE.

Note: This method will not work if the user combines a clear screen (CLS) and execution of the application into one step (`[C:\] CLS & <progrname>`), because the cursor position will be (0, 0), but the application is using the console, which belongs to CMD.EXE.

More Information:

To start a console application with its own console that will not disappear when the application is terminated, use `CMD /K`. For example, use "start `CMD /K <progrname>`".

Note that it is possible to programmatically force an application to always have its own console by immediately doing a `FreeConsole()` and an `AllocConsole()`. The disadvantage is that the C run-time handles are no longer valid. Use `CreateFile("CONIN$", ...)` with `lpSa->bInherit=TRUE`, in combination with `_open_osfhandle()` and `dup2()` to close the current handles (stdin, stdout, stderr) and associate handles that will be inherited.

Additional reference words: 3.10

INF: Detecting Windows NT from a DOS Application

Article ID: Q100290

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

There are procedures that DOS applications can do that are not supported under Windows NT. For example, calling Interrupt 25h to read the disk is not supported under Windows NT. Therefore, in some cases DOS applications will need to know whether or not they are running under Windows NT.

Interrupt 21h, function 3306h can be used by DOS applications to detect whether or not they are running under Windows NT. On return, registers BL and BH will contain the operating system major and minor numbers, respectively. If your application is running under Windows NT, the return will be:

```
BL = 5
BH = 50
```

More Information:

Note that it is important to check both BL and BH, because MS-DOS 5.0 will also return a 5 in BL.

The following code demonstrates how to detect the operating system version:

Sample Code

```
-----
#include <stdio.h>
#include <stdlib.h>
#include <io.h>

void main()
{
    unsigned char cbh = 0;
    unsigned char cbl = 0;
    unsigned char cdl = 0;
    unsigned char cdh = 0;

    _asm {
        mov ax, 3306h
        int 21h
        mov cbh, bh
        mov cbl, bl
    }
}
```



```
    printf( "After int 21h\n" );  
    printf( "%d, %d (bh, bl)\n", cbh, cbl );  
}
```

Additional reference words: 3.10

INF: Apps Should Wait to Free/Re-use WriteFileEx's Buffer
Article ID: Q90087

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The WriteFileEx function was introduced in the Win32 API. It is used to write data to a file and report its completion asynchronously (a completion routine is called when the write operation is complete). The following is the function prototype, taken from the Programmer's Reference:

```
BOOL WriteFileEx(hFile, lpBuffer, nNumberOfBytesToWrite,
                lpOverlapped, lpCompletionRoutine)
HANDLE hFile; /* file to write */
LPVOID lpBuffer; /* address of buffer */
DWORD nNumberOfBytesToWrite; /* bytes to write */
LPOVERLAPPED lpOverlapped; /* contains offset */
LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine;
```

One point that the documentation does not stress is that the data buffer, lpBuffer, should not be used by the application until the overlapped I/O operation is completed. At that time, the operating system will let the application know whether the buffer has been successfully written or whether the data transfer failed. After the status is reported, the application can free or re-use the buffer.

In general, we cannot assume that the operating system has its own copy of the data in the buffer. The kernel can transfer directly from the buffer to the device.

Additional reference words: 3.10 3.0

INF: CreateFile() Using CONOUT\$ or CONIN\$
Article ID: Q90088

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

If you attempt to open a console input or output handle by calling the CreateFile() function with the special CONIN\$ or CONOUT\$ filenames, this call will return INVALID_HANDLE_VALUE if you do not use the proper sharing attributes for the fdwShareMode parameter in your CreateFile() call. Be sure to use FILE_SHARE_READ when opening "CONIN\$" and FILE_SHARE_WRITE when opening "CONOUT\$".

INF: FlushViewOfFile() on Remote Files

Article ID: Q95043

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When flushing a memory-mapped file over a network, FlushViewOfFile() guarantees that the data has been written from the workstation, but not that the data resides on the remote disk.

This is because the server may be caching the data on the remote end. Therefore, FlushViewOfFile() may return before the data has been physically written to disk.

However, if the file is created [via CreateFile()] with FILE_FLAG_WRITE_THROUGH, the remote file system will not perform lazy writes on the file, and FlushViewOfFile() will return when the actual physical write is complete.

Additional reference words: 3.10 3.1

INF: No Way to Cancel Overlapped I/O
Article ID: Q90368

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

There is no support in Win32 to cancel an asynchronous request once it has been issued. Using CloseHandle() to close the handle, or terminating the process, does not stop the asynchronous I/O.

When a thread does an overlapped I/O (that is, a write), the system starts up another thread to do the I/O and leaves your thread free to do other work. Once it is started, there is no way to stop it.

If it necessary to interrupt the I/O, you can split the writes into batches and check for interruptions. This does not prevent the read, but you could batch the reads/writes into small pieces if you know you may want to halt it. For example, you could break a 20 megabyte (MB) write into 20, 1 MB pieces.

Additional reference words: 3.10 3.1

INF: Restriction on Named-Pipe Names

Article ID: Q100291

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

Named pipes are implemented in Windows NT using the same approach used for file systems. Within Windows NT, named pipes are represented as file objects.

During the design phase, one idea for the implementation was to allow subdirectories of named pipes. For example, a developer could create a named pipe subdirectory called `\MYPIPES`. It would then be possible to create and use pipes called `\MYPIPES\PIPE1` and `\MYPIPES\PIPE2`, but it would not be possible to use `\MYPIPES` as a pipe.

In the end, this idea was not implemented, so subdirectories are not supported. This does have some effect on the named-pipe names that are allowed. If a pipe named `\MYPIPES` is created, it is not possible to subsequently create a pipe named `\MYPIPES\PIPE1`, because `\MYPIPES` is already a pipe name and cannot be used as a subdirectory. It is possible to create a pipe named `\MYPIPES\PIPE1`, but only if there is no pipe named `\MYPIPES`.

Additional reference words: 3.10

INF: Getting Real Handle to Thread/Process Requires Two Calls
Article ID: Q90470

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The API `GetCurrentThread()` returns a pseudo-handle rather than the real handle to the thread. To get the real handle to the thread, you need to use `DuplicateHandle()` using the pseudo-handle that is returned from `GetCurrentThread()`. In addition, to get the real handle to a process, you need to call `DuplicateHandle()` after calling `GetCurrentProcess()`.

Additional reference words: 3.10 3.1

INF: Exporting Data from a DLL

Article ID: Q90530

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

It is possible for a Win32 executable to be able to address DLL global variables directly by name from within the executable. This is done by exporting global data names similar to how you would a DLL function name. Use the following steps to declare and utilize exported global data.

1. Define the global variables in the DLL code. For example:

```
int i = 1;
int *j = 2;
char *sz = "WBGLMCMTTP";
```

2. Export the variables in the module-definition (DEF) file. Use of the CONSTANT keyword is required, as shown below:

```
EXPORTS
i CONSTANT
j CONSTANT
sz CONSTANT
```

3. Declare the variables in the modules that will use them (note that they must be declared as pointers because a pointer to the variable is exported, not the variable itself):

```
extern int *i;
extern int **j;
extern char **sz;
```

4. Use the values by dereferencing the pointers declared in step 3:

```
printf( "%d", *i );
printf( "%d", **j );
printf( "%s", *sz );
```

It may simplify things to use #defines instead; then the variables can be used exactly as defined in the DLL:

```
#define i *i
#define j **j
#define sz **sz

extern int i;
extern int *j;
```



```
extern char *sz;

printf( "%d", i );
printf( "%d", *j );
printf( "%s", sz );
```

More Information:

Note: This technique can also be used to export a global variable from an application so that it can be used in a DLL.

For more information on the use of EXPORTS and CONSTANTS in the .DEF file, see Chapter 4 of the "Tools" manual.

Additional reference words: 3.10 3.1

INF: Time Stamps Under the FAT File System

Article ID: Q101186

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

Under the FAT (file allocation table) file system, Windows NT considers the time stamp on a file to be standard time if the current time is standard time, and daylight time if the current time is daylight time, regardless of what time of year the file was originally time stamped.

This is not an issue under NTFS, which consistently implements Universal Coordinated Time.

Additional reference words: 3.10

INF: Dynamic Loading of DLLs Under Windows NT
Article ID: Q90745

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When using LoadLibrary() under Win16 or OS/2, the DLL is loaded only once. Therefore, the DLL has the same address in all processes. Dynamic loading of DLLs is different under Windows NT.

A DLL is loaded separately for each process because each application has its own address space, unlike Win16 and OS/2. Pages must be mapped into the address space of a process. Therefore, it is possible that the DLL is loaded at different addresses in different processes. The memory manager optimizes the loading of DLLs so that if two processes share the same pages from the same image, they will share the same physical memory.

Each DLL has a preferred base address, specified at link time. If the address space from the base address to the base address plus image size is unavailable, then the DLL is loaded elsewhere and fixups will be applied. There is no way to specify a load address at load time.

To summarize, at load time the system:

1. Examines the image and determines its preferred base address and required size.
2. Finds the address space required and maps the image, copy-on-write, from the file.
3. Applies internal fixups if the image isn't at its preferred base.
4. Fixes up all dynamic link imports by placing the correct address for each imported function in the appropriate entry of the Import Address Table. This table is contiguous with 32-bit addresses, so 1024 imports require dirtying only one page.

More Information:

The pages containing code are shared, using a copy-on-write scheme. Copy-on-write means that the pages are read-only; however, when a process writes the page, instead of an access violation, the memory manager makes a private copy of the page and allows the write to proceed. For example, if two processes start from the same .EXE, both initially have all pages mapped from the .EXE copy-on-write. As the two processes proceed to modify pages, they get their own copies of the modified pages. The memory manager is free to optimize unmodified pages and actually map the same physical memory into the address space of both processes. Modified pages are swapped to/from the page file instead of the .EXE file.

There are two kinds of fixups. One is the address of an imported function. All these fixups are localized in what the Portable Executable specification calls the Import Address Table (IAT). This is an array of 32-bit function pointers, one for each imported API. The IAT is located on its own page(s), because it is always modified. Calling an imported function is actually an indirect call through the appropriate entry in this array. In case that the image is loaded at the preferred address, the only fixups needed are for imports.

Note that there is an optimization whereby each import library exports a 32-bit number for each API along with any name and ordinal. This serves as a "hint" to speed the fixups performed at load time. If the hints in the program and the DLL do not match, the loader uses a binary search by name.

The other kind of fixup is needed for references to the image's own code or data when the image can't be loaded at its preferred address. When a page must be taken out of memory, the system checks to see whether the page has been modified. If it has not, then the page is still mapped copy-on-write against the EXE and can be discarded from memory. Otherwise, it must be written to the page file before it can be removed from memory, so that the page file is used as the backing store (where the page is recovered from) rather than the executable image file.

Notes:

The DLL's entry point does not get called for a second LoadLibrary() call in a process (that is, no second DLL_PROCESS_ATTACH entry). There is one call to DllEntry/DLL_THREAD_ATTACH per thread no matter the number of times a thread calls LoadLibrary. The same goes for FreeLibrary(), but the DLL_THREAD_DETACH happens only on the last call (that is, reference count back to zero for the thread).

Global instance data for the DLL is on a per process basis (only one set per unique process). If it is necessary to ensure that global instance data is unique for each LoadLibrary() performed in a single process, consider thread local storage (TLS) as an alternative. This requires multiple threads of execution, but TLS allows unique data for each ThreadID. There is very little overhead on the DLL's part; just create a global TLS index during process initialization. During thread initialization, allocate memory (via HeapAlloc, GlobalAlloc, LocalAlloc, the CRT, and so on) and store a pointer to the memory using the global TLS index value in the function TlsSetValue. Win32 internally stores each thread's pointer by TLS index and ThreadID to achieve the thread specific storage.

Additional reference words: 3.10 3.0

INF: Implementing a
Article ID: Q90749

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following sample demonstrates how to implement a "kill" operation, such as a UNIX ps/kill, under Win32. Note that PSTAT.EXE gives you the PID you need.

This sample makes use of the API TerminateProcess(). While TerminateProcess() does clean up the objects owned by the process it does not notify any DLLs hooked to the process. Therefore, one can easily leave the DLL in an unstable state.

In general, the Control Panel is a much cleaner method of killing processes.

Sample Code

```
-----  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <stdarg.h>  
#include <windows.h>  
#include <string.h>  
#include <math.h>  
#include <winbase.h>  
  
void ErrorOut(char errstring[30])  
/*  
Purpose: Print out an meaningful error code by means of  
GetLastError and printf.  
  
Inputs: errstring - the action that failed, passed by the  
calling proc.  
  
Returns: none  
  
Calls: GetLastError  
*/  
  
{  
    DWORD Error;  
  
    Error= GetLastError();  
    printf("Error on %s = %d\n", errstring, Error);  
}
```

```

void main(int argc, char *argv[])
{
    HANDLE hProcess;
    DWORD ProcId;
    BOOL TermSucc;

    if (argc == 2)
    {
        sscanf(argv[1], "%x", ProcId);
        hProcess= OpenProcess(PROCESS_ALL_ACCESS, TRUE, ProcId);
        if (hProcess == NULL)
            ErrorOut("OpenProcess");
        TermSucc= TerminateProcess(hProcess, 0);
        if (TermSucc == FALSE)
            ErrorOut("TerminateProcess");
        else
            printf("Process# %.0lf terminated successfully!\n", ProcId);
    }
    else
    {
        printf("\nKills an active Process\n");
        printf("Usage: killproc ProcessID\n");
    }
}

```

Additional reference words: 3.10 3.1

PRB: try/finally with Abort() in try Body
Article ID: Q91146

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

When using the try/finally exception handling mechanism and the try body calls `abort()`, the finally body is not executed.

CAUSE

The finally body is not executed because the `abort()` never returns. It calls `ExitProcess()`, which terminates the process.

RESOLUTION

This behavior is by design.

Additional reference words: 3.10 3.1

INF: LIM-EMS Memory Limitations in a Windows NT VDM
Article ID: Q101189

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

A Windows NT virtual MS-DOS machine (VDM) session is LIM-EMS 4.0 compatible; however, each VDM is limited to 16 megabytes (MB) of memory.

A VDM is an MS-DOS session created whenever a user starts an MS-DOS application on Windows NT. Windows NT allows any number of MS-DOS applications to run simultaneously, each in its own address space.

Additional reference words: 3.10

INF: Examining the dwOemId Value

Article ID: Q101190

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The Win32 application programming interface (API) `GetSystemInfo()` fills in the members of a `SYSTEM_INFO` structure. The `dwOemId` member represents a computer identifier that is specific to a particular OEM (original equipment manufacturer). The first version of Windows NT always places a zero in the `dwOemId` member. In a later release, this behavior will change to include different OEM IDs.

Additional reference words: 3.10

INF: Win32 Priority Class Mechanism and the START Command
Article ID: Q90910

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Win32 priority class mechanism is exposed through CMD.EXE's START command.

START now accepts the following switches:

- /LOW - Start the command in the idle priority class.
- /NORMAL - Start the command in the normal priority class (this is the default).
- /HIGH - Start the command in the high priority class.
- /REALTIME - Start the command in the real-time priority class.

For a complete list of STARTs switches, type the following command at the Windows NT command prompt:

```
start /?
```

Win32 has also been modified to inherit priority class if the parent's priority class is idle; thus, a command such as

```
start /LOW nmake
```

causes build and all descendants (compiles, links, and so on) to run in the idle priority class. Use this method to do a real background build that will not interfere with anything else on your system.

A command such as

```
start /HIGH nmake
```

runs BUILD.EXE in the high priority class, but all descendants run in the normal priority class.

More Information:

Be very careful with START /HIGH and START /REALTIME. If you use either of these switches to start applications that require a lot of cycles, the applications will get all the cycles they ask for, which may cause the system to appear hung.

Additional reference words: 3.10 3.0

INF: Creating Windows in Threads

Article ID: Q90975

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In a multithreaded application, any thread can call the `CreateWindow()` API to create a window. There are no restrictions on which thread(s) can create windows.

It is important to note that the message loop and window procedure for the window must be in the thread that created the window. If a different thread creates the window, the window won't get messages from `DispatchMessage()`, but will get messages from other sources. Therefore, the window will appear but won't show activation or repaint, cannot be moved, won't receive mouse messages, and so on.

To allow threads to share input state so that they can call `SetFocus()` for a window in a different thread, it is necessary to call `AttachThreadInput()`.

Additional reference words: 3.10 3.1

INF: SHARE.EXE Functionality Built into Windows NT
Article ID: Q101191

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The functionality of the MS-DOS SHARE.EXE utility is built into the Windows NT kernel. Any application or application programming interface (API) that relies on the SHARE.EXE functionality is automatically supported. This functionality cannot be disabled.

More Information:

If you run the MS-DOS version of SHARE.EXE, you will receive a message stating that SHARE is already installed. The Windows NT MS-DOS emulation hooks Interrupt 2Fh function 10H and always returns a status indicating that SHARE is installed.

If you run an MS-DOS application and it complains that SHARE.EXE is not installed, the application may be searching the AUTOEXEC.BAT file for a "share" string rather than using the proper Interrupt 2Fh interface.

Additional reference words: 3.10

PRB: try/finally with return in finally Body Preempts Unwind
Article ID: Q91147

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

Returning out of a finally returns from the containing procedure scope. For instance, in the following code fragment, the return in the finally block results in a return from func():

```
int func()
{
    int status = 0;
    __try {
        ...
        status = test();
        ...
    }
    __finally {
        if (status != 0) {
            status = FAILURE;
            return status;
        }
    }
    return status;
}
```

CAUSE

A return from within a __finally is equivalent to a goto to the closing brace in the enclosing function [for example, func()]. This is allowed, but has consequences that should normally be avoided.

Exception handling has two stages. First, the exception stack is walked, looking for an accepting __except. When an accepting handler has been found, all __finallys between the top-of-exception-stack and the target __except will be called. During this "unwind", the __finallys are assumed to each execute and then return to their caller (the system unwind code).

A return in a __finally abnormally aborts this unwinding. Instead of returning to the system unwinder, the __finally returns to the enclosing function's caller [for example, func()'s parent]. The accepting __except filter may set some status or perform an allocation in anticipation of the __except handler being entered. In this case, the intervening __finally with the return will stop the unwind, and the __except handler is never entered.

RESOLUTION

This is by design. It makes it possible for a finally handler to stop an unwind and return a status. This is what is referred to as a collided unwind.

Abnormal termination from try/except or try/finally blocks is not generally recommended because it is a performance hit.

The example can be rewritten so that the unwind chain is not aborted:

```
int func()
{
    int status = 0;
    __try {
        ...
        status = test();
        ...
    }
    __except(status != 0) {

        /* null */
    }
    if (status != 0)
        status = FAILURE;
    return status;
}
```

This does not have identical semantics because the exception filters higher up the exception stack will not be executed. However, ensuring that both phases of exception handling progress to the same depth is a more robust solution.

More Information:

Normally this behavior is transparent to any higher-level exception handling code. If, however, a filter function, as a side effect, stores information that it expects to process in an exception handler, then it may or may not be transparent. Storing such information in a filter function should be avoided because it is always possible that the exception handler will not be executed because the unwind is preempted. In the absence of storing such side effects, it will be transparent that an exception occurred and an attempted unwind occurred if one of the descendent functions has a try/finally block with a finally clause that preempts the unwind.

Additional reference words: 3.10 3.1

INF: Initiating an Unwind in an Exception Handler
Article ID: Q91148

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In the "Programmer's Reference: Overviews," in the chapter titled "Structured Exception Handling," the last paragraph of the Exception Handling and Unwind subsection says that "the exception handler usually initiates an unwind operation by calling a system supplied function".

The paragraph describes what the compiler-generated code for try-except does. Initiating the unwind is automatically handled by the compiler when you exit the handler in any way other than by sequentially executing the final statement, for example, with a goto, return, break, and so forth. See the example below.

Because exception handling can be quite different under different processor architectures, the description uses words such as "usually"; "usually" you initiate an unwind. In some cases (for example, a goto to a label in the current call frame), you don't need to unwind (you already did when you hit the handler), so none is needed, and the compiler need not bother calling the system to do one.

Note that unwinds aren't a fast operation. It would generally be better for the exception filter to decide the exception can't be handled, and evaluate to EXCEPTION_CONTINUE_SEARCH, than to force the unwind this way. If some cleanup still needs doing, the termination handler will be called during the unwind operation and could do the cleanup.

More Information:

The following code fragment demonstrates initiating an unwind by exiting the exception handler with a goto statement:

```
int func() {  
  
    try {  
  
        ThisCode( BuggyParameter );  
  
    }  
  
    except( EXCEPTION_EXECUTE_HANDLER ) {  
  
        if( ICannotContinue( BuggyParameter ) )  
            goto InitUnwind; // <- This initiates the desired unwind  
  
        HandleProblemBrilliantly( BuggyParameter );  
  
    }  
}
```



```
    }  
  
InitUnwind:  
    ProcessData();  
    IssueError();  
    RetireGracefully();  
}
```

Additional reference words: 3.10 3.1

INF: Using volatile to Prevent Optimization of try/finally
Article ID: Q91149

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following is an example of a valid optimization that may take programmers by surprise.

1. A variable (temp) used only within the try-except body is declared outside it, and therefore is global with respect to the try.
2. Assignment to the variable (temp) is in the program only for a possible side effect of doing a read memory access through the pointer.

For example:

```
VOID
puRoutine( PULONG pu )
{
    ...
    ULONG temp;        // Just for probing
    ...
    try {
        temp = *pu;    // See if pu is a valid argument
    }

    except {
        // Handle exception
    }
}
```

The compiler optimizes and eliminates the entire try-except statement because temp is not used later.

If the value of temp were used globally, the compiler should treat the assignment to temp as volatile and do the assignment immediately even if it is overwritten later in the body of the try. The reasoning is that, at almost any point in the try body, control may jump to the except (or an exception filter). Presumably the programmer accessing the variable outside the try wants to get the current (most recently assigned) value.

The way to prevent the compiler from performing the optimization is:

```
temp = (volatile ULONG) *pu;
```

If a temporary variable is not needed, given the example, the read access should still be specified as volatile, for example:

*(volatile PULONG) pu;

Additional reference words: 3.10

INF: Icons for Console Applications

Article ID: Q91150

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Under OS/2, when adding an application named CONAPP.EXE to a program group, the system uses the file CONAPP.ICO (if it exists) as the icon. This does not happen automatically under Windows NT; the item will have a generic icon.

To specify the icon that appears in the program group, use the following steps:

1. Create a resource file containing an ICON statement:

```
ConApp ICON ConApp.ICO
```

2. Compile the resource using RC:

```
rc -r -fo conapp.res $(cvars) conapp.rc
```

3. Use CvtRes to produce an .RBJ file:

```
cvtres -$(CPU) conapp.res -o conapp.rbj
```

4. Include the .RBJ file in the link command.

Steps 2-4 are intended for use in a makefile that does an !include of NTWIN32.MAK. See the makefiles for the SDK samples for more detail.

More Information:

If the application is started by clicking its icon in Program Manager, the icon that appears when the application is minimized will be that icon, whether it is a generic icon or an icon imbedded in the executable file.

If the application is started from the MS-DOS prompt or the File menu, then the icon that appears when the application is minimized will be the icon that is used for the MS-DOS prompt.

Additional reference words: 3.10 3.1

PRB: Maximum Memory Handles

Article ID: Q91194

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following limitations exist when allocating memory handles. The minimum block you can allocate with each call to `VirtualAlloc()` is 64K. There is a per-process limit on the number of handles you can allocate. If you allocate 32K handles, you run out of the 2GB user address space. When calling `HeapAlloc()`, there is no limit to the number of handles. Microsoft has successfully allocated over 1,000,000 handles on a growable heap using `HeapAlloc()`. `GlobalAlloc()` and `LocalAlloc()` (combined) are limited to 65536 total handles for `GMEM_MOVEABLE` and `LMEM_MOVEABLE` memory per process. Note that this limitation does not apply to `GMEM_FIXED` or `LMEM_FIXED` memory.

Additional reference words: 3.10 3.1

INF: PAGE_READONLY May Be Used as Discardable Memory

Article ID: Q94947

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Virtual memory pages marked as PAGE_READONLY under Win32 may be used the way discardable segments of memory are used in Windows 3.1. These virtual memory pages are by default not "dirty," so the system may use them (zeroing them first if necessary) without having to first write their contents to disk.

From a system resource perspective, PAGE_READONLY is treated similar to discardable memory under Windows 3.1 when the system needs to free up resources. From a programming standpoint, the system automatically re-reads the memory when the data is next accessed (for example, we attempt to access our page when it has been "discarded," a page fault is generated, and the system reads it back in). Memory-mapped files are convenient for setting up this type of behavior.

If a PAGE_READONLY memory page becomes dirty [by changing the protection via VirtualProtect() to PAGE_READWRITE, changing the data, and restoring PAGE_READONLY], the page will be written to disk before the system uses it.

Additional reference words: 3.10 3.1

PRB: Return Values of Performance APIs

Article ID: Q91215

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In the Windows Help file for the Win32 API, the APIs QueryPerformanceCounter() and QueryPerformanceFrequency() are incorrectly documented as returning FALSE when a high performance timer is not present and TRUE when a high performance timer is present.

The reverse is correct; FALSE (SUCCESS) is returned when a high performance counter is present in the system and TRUE (!SUCCESS) is returned when a high performance counter is not present.

For x86 and R4000 systems, FALSE (SUCCESS) is always returned because high performance counters are available on both platforms.

Additional reference words: 3.10 3.1

INF: Sharing Win32 Services

Article ID: Q91698

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Win32 services are discussed in the overview for the Service Control Manager. The documentation says that:

A Win32 service runs in a Win32 process which it may or may not share with other Win32 services.

Whether or not a service has its own process is determined by which of these service types is specified in the call to CreateService() to add the service to the Service Control Manager Database.

SERVICE_WIN32_OWN_PROCESS

This service type indicates that only one service can run in the process. This allows an application to spawn multiple copies of a service under different names, each of which gets its own process. This is the most common type of service.

SERVICE_WIN32_SHARE_PROCESS

This service type indicates that more than one service can be run in a single process. When the second service is started, it is started as a thread in the existing process. A new process is not created. An example of this is the LAN Manage Workstation and the LAN Manager Server. Note that the service must be started in the system account, which is .\System. The name must be NULL.

The service type for each service is stored in the registry. The value is 0x10 for SERVICE_WIN32_OWN_PROCESS or 0x20 for SERVICE_WIN32_SHARE_PROCESS.

Additional reference words: 3.1 3.10

INF: Determining Whether Windows NT Is Running
Article ID: Q92395

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Programmers can use `GetVersion()` to determine what version of Windows their applications are running under. This might be a version of MS-DOS/Windows, Windows on Windows (WOW), MS-DOS/Windows using extensions (Win32s), Windows NT, or Windows NT Advanced Server. According to the documentation, the return value of `GetVersion()` is a `DWORD` that specifies the major and minor version numbers.

The following table shows the return values from `GetVersion()` under various environments:

Environment	LOWORD	HIWORD
Win 3.x	Windows version	MS-DOS version
WOW	Windows version 3.1	MS-DOS version 5.0
Win32s	Windows version 3.1	RESERVED *
Win32	Windows version 3.1	RESERVED **

* The highest bit is 1. Note that the version of MS-DOS cannot be determined as it can under Windows 3.x.

** The highest bit is 0. The remaining bits specify build number.

Note that Windows 3.1 and WOW can return the same results. Therefore, `GetVersion()` is especially useful only for 32-bit applications to determine whether they are running under Windows NT or under Windows 3.1 with Win32s. This is done by masking off the high bit.

Therefore, 16-bit applications should use `GetWinFlags()` to determine whether they are being run under MS-DOS/Windows or WOW. `GetWinFlags()` returns a `WF_WINNT` flag if the application is running under WOW.

`GetWinFlags()` is an existing function that has had these the following flag added in WOW:

```
#define WF_WINNT          0x4000
```

`GetWinFlags()` is not a part of the Win32 API. Processor information can be found through the new Win32 API `GetSystemInfo()`.

In order to distinguish between Windows NT and Windows NT Advanced Server, use the registry API to query the following:

```
\HKEY_LOCAL_MACHINE\SYSTEM  
\CurrentControlSet
```

\Control
\ProductOptions

The result will be either WINNT or LANMANNT (for NTAS).

Additional reference words: 3.10 3.1

INF: Interrupting Threads in Critical Sections

Article ID: Q101193

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

If a thread enters a critical section and then terminates abnormally, the critical section object will not be released. Many components of the C run-time library are not reentrant and use a resource locking scheme to maintain coherency in the multithreaded environment. Thus, a thread that has entered a C run-time function, such as `printf()`, could deadlock all access (within that process) to `printf()` if it terminates abnormally.

This situation could arise if a thread is terminated with `TerminateThread()` while it holds a resource lock. If this occurs, any thread that tries to acquire that resource lock will become deadlocked.

Additional reference words: 3.10

INF: New DLL: LOCALMON.DLL

Article ID: Q92507

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The new DLL, LOCALMON.DLL, resides in WINNT\SYSTEM32. This new DLL is a monitor that manages ports at a high level.

Monitors are responsible for sending the print job to a port. If the port is a communications port, the monitor will open the port (COM1), set the baud rate, parity, and so on, as well as write the data and close the port.

Monitors allow users to print to a LaserJet IIISi, to PostScript printers on an AppleTalk network, or to named pipes. Monitor management and UI can be configured within the Print Manager by selecting the port from a combo box.

Additional reference words: 3.10 3.1

INF: Changes to DLL Makefiles Made for Final Release

Article ID: Q101337

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The documentation and sample makefiles for beta versions of the Win32 software development kit (SDK) use a different entry point for dynamic-link libraries (DLLs) than is recommended for the final release. This difference is important to note if your DLLs are linked with the C run time.

Previously, Microsoft recommended calling `_crt_init()` in your DLL entry point (or making it the DLL entry point), so that the C run time would be correctly initialized. Currently, Microsoft recommends that all DLLs provide a main function

```
BOOL WINAPI DllMain( HANDLE hDLL, DWORD dwReason, LPVOID lpReserved )
```

and that the entry point be specified with the following linker option:

```
-entry:_DllMainCRTStartup$(DLENTY)
```

More Information:

The loader calls `DllMainCRTStartup()`, which is provided in the C run-time library. This routine handles all C run time and C++ initialization, then determines whether `DllMain()` is exported from the DLL, and executes it if it exists.

Additional reference words: 3.10

**INF: Impersonation Provided by ImpersonateNamedPipeClient()
Article ID: Q101378**

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The following information is from the Win32 application programming interface (API) "Programmer's Reference" in the section regarding ImpersonateNamedPipeClient():

```
BOOL ImpersonateNamedPipeClient(HANDLE hNamedPipe)
```

The ImpersonateNamedPipeClient function impersonates a named-pipe client application.

The level of impersonation can be specified by the client when the named pipe is opened. If the client does not explicitly specify a level, then the default is SecurityImpersonation.

More Information:

Suppose there are three threads (A, B, and C) where:

A calls B

B calls C

B does a SecurityImpersonation of A

If A and B both specify dynamic tracking, then C can see the context of A when it makes a call on the pipe, as long as B impersonates A. Otherwise, C will see the context of A only if B was impersonating A when the pipe between B and C was connected.

Note that dynamic tracking is not supported between machines.

Additional reference words: 3.10

INF: Distributed Computing Environment (DCE) Compliance
Article ID: Q92515

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Remote Procedure Call (RPC) is one part of the OSF Distributed Computing Environment (DCE) model. The following background information comes from the introduction to RPC in the "Remote Procedure Call Programmer's Guide and Reference":

The design and technology behind Microsoft RPC is just one part of a complete environment for distributed computing defined by the Open Software Foundation (OSF), a consortium of companies formed to define the components of a complete environment that supports distributed computing. The OSF requests proposals for standards, accepts comments on the proposals, votes whether to accept the standards and promulgates them.

Generally speaking, Microsoft has tested interoperability with DCE connection-oriented RPC 1.0.1. We are DCE connection-oriented RPC compliant, not compliant with the whole DCE. For example, there is no support for the OSF DCE Core Services (that is, Directory Service, Security Service, Time Service, and so on) or the OSF DCE Extended Services (that is, Distributed File Service, Diskless Support Service, Personal Computer Integration Service, and so on).

More Information:

Microsoft RPC, which interoperates with connection-oriented DCE RPC, is supported on MS-DOS, Windows, and Windows NT. MS-DOS and Windows support RPC clients. Windows NT supports both RPC clients and servers.

DCE has been certified using TCP/IP and UDP/IP. Microsoft RPC under MS-DOS supports LAN Manager version 2.1 TCP only. Under Windows, TCP is supported from any vendor that is compliant with the Windows Sockets specification. Under Windows NT, we support the TCP packaged in the product. In addition to those protocols, which are of interest to people looking at interoperability, Microsoft RPC also supports NetBIOS and Named Pipes.

The OSF will be sponsoring inter-vendor tests early in 1993. Microsoft intends to attend. At the current time, DEC is the only vendor shipping DCE. Microsoft has tested interoperability with Ultrix and VMS machines.

Microsoft RPC is part of the Win32 SDK. The customer has the right to freely distribute run times with applications if the run time is not already distributed with the operating system.

Additional reference words: 3.10 3.1

INF: Process Will Not Terminate Unless System Is In User-mode
Article ID: Q92761

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Under Windows NT, a process will not be terminated unless the system is in user-mode. Suppose that `TerminateProcess()` is called while a device driver or filesystem code is being executed. The system will wait until the threads are running user code before marking the process for termination. On system exit, processes that were the target of a `TerminateProcess()` will be killed.

This may affect drivers. If a driver is waiting for an object or multiple objects in `WaitMode` or `UserMode`, its wait may complete unsuccessfully due to a termination request. Any code that does a `UserMode` wait or an `Alertable` wait must check the return status of the wait call. If the wait fails with `STATUS_USER_APC` or `STATUS_ALERTED`, this is not an error. The driver should cleanup and return to user-mode.

Additional reference words: 3.10

INF: Non-Address Range in Address Space

Article ID: Q92764

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Each process has its own private address space. The process can use up to 2 gigabytes of virtual memory. This 2Gb is not necessarily contiguous. The system uses the other 2Gb.

The user-mode addresses extend from 0x00010000 to 0x7FFF0000. The following ranges are reserved as non-address space to ensure that the process does not walk on system-owned memory

0x00000000 to 0x0000FFFF (first 64K of virtual space)

0x7FFF0000 to 0x7FFFFFFF (last 64K of user virtual space)

These are effectively PAGE_NOACCESS ranges.

Additionally, Win32 DLLs will reserve other specific address ranges. For more information, see the file COFFBASE.TXT that comes with the DDK.

More Information:

This range is not guaranteed to serve this purpose in the future. There could be good reasons in a future implementation to use these addresses. Code that is going to depend on this non-address range should verify its validity at run time with something like

```
BOOL IsFirst64kInvalid(void)
{
    BOOL bFirst64kInvalid = FALSE;

    try {
        *(char *)0x0000FFFF;
    }
    except (EXCEPTION_EXECUTE_HANDLER) {
        if (EXCEPTION_ACCESS_VIOLATION == GetExceptionCode())
            bFirst64kInvalid = TRUE;
    }

    return bFirst64kInvalid;
}
```

INF: Alternatives to Using GetProcAddress With LoadLibrary
Article ID: Q92862

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When loading a DLL with `LoadLibrary()`, an alternative to calling `GetProcAddress()` for each of your DLL entry points is to have the DLL initialization function initialize a global structure or array containing the addresses of these DLL entry points, then call a DLL function from your executable which will return the address of this structure or array to your executable. You can then call your DLL functions via the function pointers in this structure or array.

The best place to initialize this structure or array of function pointers would be in the `DLL_PROCESS_ATTACH` code of your DLL's main entry point. The structure or array containing these function pointers must be declared as either a global variable or as dynamically allocated memory (`malloc()`, `GlobalAlloc()`, etc.) in your DLL in order for the executable to be able to address this memory properly.

It is also possible, though not as clean, to export the global structure or array of function pointers so that your executable can use the structure or array by name directly in your executable. For more information on how to declare and export global data in a Win32 DLL query on the following words in this Knowledge Base:

global and exported and data and DLL

Be careful not to call these DLL functions via the function pointers after the DLL is unloaded via `FreeLibrary()`. After `FreeLibrary()` is called, these function pointer addresses are invalid and calling them will result in an access violation.

This technique of returning pointers to DLL entry points is a supported technique and will work on all hardware platforms that Windows NT supports.

INF: Gaining Access to ACLs

Article ID: Q102098

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

To gain access to a security access control list (SACL), a process must have the SE_SECURITY_NAME privilege. When requesting access, the calling process must request ACCESS_SYSTEM_SECURITY in the desired access mask.

There is not a privilege that controls read or write access to a discretionary access control list (DACL). Instead, access to read and write an object's DACL is granted by the READ_CONTROL and WRITE_DAC access rights, respectively. These rights must be specifically granted to the user (or group containing the user) for DACL read or write access to be granted. If the owner of an object requests READ_CONTROL or WRITE_DAC, the access will always be granted.

Additional reference words: 3.10

INF: Maximum GlobalAddAtom() String Size Is 32K Characters
Article ID: Q94951

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Under Win32, the maximum string length for the lpszName parameter of GlobalAddAtom() is 32K characters (64K for Unicode) and/or available memory.

Additional reference words: 3.10 3.0

INF: Administrator Access to Files

Article ID: Q102099

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

A user that is a member of the Administrator group is not automatically granted access to any file on the local machine. For an administrator to access a file, permission must be specifically granted (as for any user) in the file's discretionary access control list (DACL).

If an administrator wants to access a file that he or she is not granted access to, the administrator must first take ownership of that file. Once ownership is taken, the administrator will have full access to the file. It is important to note that administrator cannot give ownership back to the original owner. If this were so, the administrator could take ownership of a file, examine it, and then assign it back to the original owner without that owner's knowledge.

NOTE: Because administrators have backup privileges, an administrator could back up a file (or entire volume) and restore it onto another system. The administrator could then take ownership of a file on this new system without the owner's knowledge. Please keep this in mind when thinking about file security.

Additional reference words: 3.10

INF: Passing Security Information to SetFileSecurity()

Article ID: Q102100

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The SetFileSecurity() Win32 application programming interface (API) takes a pointer to a Security Descriptor. This is because SetFileSecurity() can set any of the following security information for a file:

- The owner identifier of the file
- The primary group identifier of the file
- The discretionary access-control list (DACL) of the file
- The system access-control list (SACL) of the file

When you pass the SD and SECURITY_INFORMATION structure to SetFileSecurity(), the SECURITY_INFORMATION structure identifies which security information is to be set. The SECURITY_INFORMATION structure is a DWORD that can be one of the following values:

- OWNER_SECURITY_INFORMATION
- GROUP_SECURITY_INFORMATION
- DACL_SECURITY_INFORMATION
- SACL_SECURITY_INFORMATION

Each of these values represents one of the security items listed above. The SD that is passed to SetFileSecurity() is simply a container for the security information being set for the specified file. SetFileSecurity() examines the value in the SECURITY_INFORMATION structure, extracts the appropriate information from the provided SD, and applies it to the specified file's SD.

Additional reference words: 3.10

INF: Extracting the SID from an ACE

Article ID: Q102101

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

To access the security identifier (SID) contained in an access control entry (ACE), the following syntax can be used:

```
PSID pSID;

if(((PACE_HEADER)pTempAce)->AceType) == ACCESS_ALLOWED_ACE_TYPE)
{
    pSID=(PSID)&((PACCESS_ALLOWED_ACE)pTempAce)->SidStart;
}
```

The "if" statement checks the type of ACE, which is one of the following values:

```
ACCESS_ALLOWED_ACE_TYPE
ACCESS_DENIED_ACE_TYPE
SYSTEM_AUDIT_ACE_TYPE
```

The conditional statement casts pTempAce (the pointer to the ACE) to a PACCESS_ALLOWED_ACE structure. The address of the SidStart member is then cast to a PSID and assigned to the pSID variable. pSID can now be used with any Win32 Security application programming interface (API) that takes a PSID as a parameter.

Additional reference words: 3.10

INF: How to Add an Access-Allowed ACE to a File

Article ID: Q102102

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

This article explains the process of adding an access-allowed (or access-denied) access control entry (ACE) to a file.

Adding an access-allowed ACE to a file's access control list (ACL) provides a means of granting or denying (using an access-denied ACE) access to the file to a particular user or group. In most cases, the file's ACL will not have enough free space to add an additional ACE, and therefore it is usually necessary to create a new ACL and copy the file's existing ACEs over to it. Once the ACEs are copied over and the access-allowed ACE is also added, the new ACL can be applied to the file's security descriptor (SD). This process is explained in detail in the section below. Sample code is provided at the end of this article.

MORE INFORMATION

=====

At the end of this article is sample code that defines a function named `AddAccessRights()`, which adds an access-allowed ACE to the specified file allowing the specified access. Steps 1-17 in the comments of the sample code are discussed in detail below:

1. `GetUserName()` is called to retrieve the name of the currently logged in user. The user name is stored in `plszUserName[]` array.
2. `LookupAccountName()` is called to obtain the SID of the user returned by `GetUserName()` in step 1. The resulting SID is stored in the `UserSID` variable and will be used later in the `AddAccessAllowedACE()` application programming interface (API) call. The `LookupAccountName()` API is also providing the user's domain in the `plszDomain[]` array. Please note that `LookupAccountName()` returns the SID of the first user or group that matches the name in `plszUserName`.
3. `GetFileSecurity()` is used here to obtain a copy of the file's security descriptor (SD). The file's SD is placed into the `ucSDbuf` variable, which is declared a size of `65536+SECURITY_DESCRIPTOR_MIN_LENGTH` for simplicity. This value represents the maximum size of an SD, which ensures the SD will be of sufficient size.
4. Here we initialize the new security descriptor (`NewSD` variable) by

calling the `InitializeSecurityDescriptor()` API. Because the `SetFileSecurity()` API requires that the security item being set is contained in a SD, we create and initialize `NewSD`.

5. Here `GetSecurityDescriptorDacl()` retrieves a pointer to the discretionary access control list (DACL) in the SD. The pointer is stored in the `pACL` variable.
6. `GetAclInformation()` is called here to obtain size information on the file's DACL in the form of a `ACL_SIZE_INFORMATION` structure. This information is used when computing the size of the new DACL and when copying ACEs.
7. This statement computes the exact number of bytes to allocate for the new DACL. The `AclBytesInUse` member represents the number of bytes being used in the file's DACL. We add this number to the size of an `ACCESS_ALLOWED_ACE` and the size of the user's SID. Subtracting the size of a `DWORD` is an adjustment required to obtain the exact number of bytes necessary.
8. Here we allocate memory for the new ACL that will ultimately contain the file's existing ACEs plus the access-allowed ACE.
9. In addition to allocating the memory, it is important to initialize the ACL structure as we do here.
10. Here we check the `bDaclPresent` flag returned by `GetSecurityDescriptorDacl()` to see if a DACL was present in the file's SD. If a DACL was not present, then we skip the code that copies the file's ACEs to the new DACL.
11. After verifying that there is at least one ACE in the file's DACL (by checking the `AceCount` member), we begin the loop to copy the individual ACEs to the new DACL.
12. Here we get a pointer to an ACE in the file's DACL by using the `GetAce()` API.
13. Now we add the ACE to the new DACL. It is important to note that we pass `MAXDWORD` for the `dwStartingAceIndex` parameter of `AddAce()` to ensure the ACE is added to the end of the DACL. The statement `((PACE_HEADER)pTempAce)->AceSize` provides the size of the ACE.
14. Now that we have copied all the file's original ACEs over to our new DACL, we add the access-allowed ACE. The `dwAccessMask` variable will contain the access mask being granted. `GENERIC_READ` is an example of an access mask.
15. Because the `SetFileSecurity()` API can set a variety of security information, it takes a pointer to a security descriptor. For this reason, it is necessary to attach our new DACL to a temporary SD. This is done by using the `SetSecurityDescriptorDacl()` API.
16. Now that we have a SD containing the new DACL for the file, we set the DACL to the file's SD by calling `SetFileSecurity()`. The `DACL_SECURITY_INFORMATION` parameter indicates that we want the DACL in the provided SD applied to the file's SD. Please note that

only the file's DACL is set, the other security information in the file's SD remains unchanged.

17. Here we free the memory that was allocated for the new DACL.

The below sample demonstrates the basic steps required to add an access-allowed ACE to a file's DACL. Please note that this same process can be used to add an access-denied ACE to a file's DACL. Because the access-denied ACE should appear before access-allowed ACEs, it is suggested that the call to `AddAccessDeniedAce()` precede the code that copies the existing ACEs to the new DACL.

Sample Code

```
#define SD_SIZE (65536 + SECURITY_DESCRIPTOR_MIN_LENGTH)

BOOL AddAccessRights(CHAR *pFileName, DWORD dwAccessMask)
{
    // SID variables

    UCHAR          psnuType[2048];
    UCHAR          lpszDomain[2048];
    DWORD          dwDomainLength = 250;
    UCHAR          UserSID[1024];
    DWORD          dwSIDBufSize=1024;

    // User name variables

    UCHAR          lpszUserName[250];
    DWORD          dwUserNameLength = 250;

    // File SD variables

    UCHAR          ucSDbuf[SD_SIZE];
    PSECURITY_DESCRIPTOR pFileSD=(PSECURITY_DESCRIPTOR)ucSDbuf;
    DWORD          dwSDLengthNeeded;

    // ACL variables

    PACL          pACL;
    BOOL          bDaclPresent;
    BOOL          bDaclDefaulted;
    ACL_SIZE_INFORMATION AclInfo;

    // New ACL variables

    PACL          pNewACL;
    DWORD          dwNewACLSize;

    // New SD variables

    UCHAR          NewSD[SECURITY_DESCRIPTOR_MIN_LENGTH];
    PSECURITY_DESCRIPTOR psdNewSD=(PSECURITY_DESCRIPTOR)NewSD;

    // Temporary ACE
```

```

PVOID          pTempAce;
UINT           CurrentAceIndex;

// STEP 1: Get the logged on user name

if(!GetUserName(lpszUserName,&dwUserNameLength))
{
    printf("Error %d:GetUserName\n",GetLastError());
    return(FALSE);
}

// STEP 2: Get SID for current user

if (!LookupAccountName((LPSTR) NULL,
    lpszUserName,
    UserSID,
    &dwSIDBufSize,
    lpszDomain,
    &dwDomainLength,
    (PSID_NAME_USE)psnuType))
{
    printf("Error %d:LookupAccountName\n",GetLastError());
    return(FALSE);
}

// STEP 3: Get security descriptor (SD) for file

if(!GetFileSecurity(pFileName,
    (SECURITY_INFORMATION) (DACL_SECURITY_INFORMATION),
    pFileSD,
    SD_SIZE,
    (LPDWORD) &dwSDLengthNeeded))
{
    printf("Error %d:GetFileSecurity\n",GetLastError());
    return(FALSE);
}

// STEP 4: Initialize new SD

if(!InitializeSecurityDescriptor(psdNewSD,SECURITY_DESCRIPTOR_REVISION))
{
    printf("Error %d:InitializeSecurityDescriptor\n",GetLastError());
    return(FALSE);
}

// STEP 5: Get DACL from SD

if (!GetSecurityDescriptorDacl(pFileSD,
    &bDaclPresent,
    &pACL,
    &bDaclDefaulted))
{
    printf("Error %d:GetSecurityDescriptorDacl\n",GetLastError());
    return(FALSE);
}

```

```

// STEP 6: Get file ACL size information

if(!GetAclInformation(pACL, &AclInfo, sizeof(ACL_SIZE_INFORMATION),
    AclSizeInformation))
{
    printf("Error %d:GetAclInformation\n", GetLastError());
    return(FALSE);
}

// STEP 7: Compute size needed for the new ACL

dwNewACLSize = AclInfo.AclBytesInUse +
    sizeof(ACCESS_ALLOWED_ACE) +
    GetLengthSid(UserSID) - sizeof(DWORD);

// STEP 8: Allocate memory for new ACL

pNewACL = (PACL)LocalAlloc(LPTR, dwNewACLSize);

// STEP 9: Initialize the new ACL

if(!InitializeAcl(pNewACL, dwNewACLSize, ACL_REVISION2))
{
    printf("Error %d:InitializeAcl\n", GetLastError());
    LocalFree((HLOCAL) pNewACL);
    return(FALSE);
}

// STEP 10: If DACL is present, copy it to a new DACL

if(bDaclPresent) // only copy if DACL was present
{
    // STEP 11: Copy the file's ACEs to our new ACL

    if(AclInfo.AceCount)
    {
        for(CurrentAceIndex = 0; CurrentAceIndex < AclInfo.AceCount;
            CurrentAceIndex++)
        {
            // STEP 12: Get an ACE

            if(!GetAce(pACL, CurrentAceIndex, &pTempAce))
            {
                printf("Error %d: GetAce\n", GetLastError());
                LocalFree((HLOCAL) pNewACL);
                return(FALSE);
            }

            // STEP 13: Add the ACE to the new ACL

            if(!AddAce(pNewACL, ACL_REVISION, MAXDWORD, pTempAce,
                ((PACE_HEADER)pTempAce)->AceSize))
            {
                printf("Error %d:AddAce\n", GetLastError());
                LocalFree((HLOCAL) pNewACL);
                return(FALSE);
            }
        }
    }
}

```

```

    }
}

// STEP 14: Add the access-allowed ACE to the new DACL

if(!AddAccessAllowedAce(pNewACL,ACL_REVISION2,dwAccessMask, &UserSID))
{
    printf("Error %d:AddAccessAllowedAce",GetLastError());
    LocalFree((HLOCAL) pNewACL);
    return(FALSE);
}

// STEP 15: Set our new DACL to the file SD

if (!SetSecurityDescriptorDacl(psdNewSD,
    TRUE,
    pNewACL,
    FALSE))
{
    printf("Error %d:SetSecurityDescriptorDacl",GetLastError());
    LocalFree((HLOCAL) pNewACL);
    return(FALSE);
}

// STEP 16: Set the SD to the File

if (!SetFileSecurity(pFileName, DACL_SECURITY_INFORMATION,psdNewSD))
{
    printf("Error %d:SetFileSecurity\n",GetLastError());
    LocalFree((HLOCAL) pNewACL);
    return(FALSE);
}

// STEP 17: Free the memory allocated for the new ACL

LocalFree((HLOCAL) pNewACL);
return(TRUE);
}

```

Additional reference words: 3.10

INF: Computing the Size of a New ACL
Article ID: Q102103

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

When adding an access-allowed access control entry (ACE) to a discretionary access control list (DACL), it is useful to know the exact size needed for the new DACL. This is particularly useful when creating a new DACL and copying over the existing ACEs. The below code computes the size needed for a DACL if an access-allowed ACE is added:

```
ACL_SIZE_INFORMATION AclInfo;

GetAclInformation(pACL, &AclInfo, sizeof(ACL_SIZE_INFORMATION),
                AclSizeInformation)

dwNewACLSize = AclInfo.AclBytesInUse +
                sizeof(ACCESS_ALLOWED_ACE) +
                GetLengthSid(UserSID) - sizeof(DWORD);
```

The call to `GetAclInformation()` takes a pointer to an ACL. This point is supplied by your program and should point to the DACL you want to add an access-allowed ACE to. The `GetAclInformation()` call fills out a `ACL_SIZE_INFORMATION` structure, which provides size information on the ACL.

The second statement computes what the new size of the ACL will be if an access-allowed ACE is added. This is accomplished by adding the current bytes being used to the size of an `ACCESS_ALLOWED_ACE`. We then add the size of the security identifier (SID) (provided by your application) that is to be used in the `AddAccessAllowedAce()` API call. Subtracting out the size of a `DWORD` is the final adjustment needed to obtain the exact size. This adjust is to compensate for a place holder member in the `ACCESS_ALLOWED_ACE` structure which is used in variable length ACEs.

When adding an ACE to an existing ACL, often there is not enough free space in the ACL to accommodate the additional ACE. In this situation, it is necessary to allocate a new ACL and copy over the existing ACEs and then add the access-allowed ACE. The above code can be used to determine the amount of memory to allocate for the new ACL.

Additional reference words: 3.10

PRB: Determining Whether App Is Running as Service or .EXE
Article ID: Q94994

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When debugging Windows NT services, it may be necessary for the service application to run interactively.

In this case, the application may determine whether it is being run as a service or as an executable (interactively) by checking GetLastError() after the call to StartServiceCtrlDispatcher() in the application's startup code.

If the application is being run as an executable, the call to GetLastError() will return with the following:

```
ERROR_FAILED_SERVICE_CONTROLLER_CONNECT
```

More Information:

Sample Code

```
-----  
  
// Call StartServiceCtrlDispatcher() to set up the control  
// interface. The API won't return until all services have been  
// terminated. At that point, we just exit. See the  
// StartServiceCtrlDispatcher() entry in Windows Help.  
  
if (!StartServiceCtrlDispatcherW(ElfSvcDispatchTable) &&  
    GetLastError() == ERROR_FAILED_SERVICE_CONTROLLER_CONNECT) {  
// Set a flag indicating you're running as an .EXE, not a service.  
}
```

Additional reference words: 3.10 3.1

INF: VirtualLock() Only Locks Pages into Working Set
Article ID: Q94996

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

VirtualLock() locks pages only into an application's working set; it does not lock them absolutely into memory.

VirtualLock() lock pages into virtual memory, and therefore as long as a given process is in real (physical) memory, the virtually locked page is guaranteed to be in real memory as well. VirtualLock() essentially means "this page is always part of my process's working set."

However, the system is free to swap out any virtually locked pages if it swaps out the whole process. And when the system swaps the process back in, the virtually locked pages (similar to any virtual pages) may end up residing in different real pages.

It is wise to use VirtualLock() very sparingly because it reduces the flexibility of the system. Depending upon memory demands on the system, the memory manager may vary the number of pages a process can lock. Under typical conditions you can expect to be able to VirtualLock() approximately 28 to 32 pages.

Additional reference words: 3.10 3.1

INF: Trapping Floating-Point Exceptions Under Windows NT
Article ID: Q94998

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The `_controlfp()` function is the portable equivalent to the `_control87()` function, which is documented in the Microsoft C/C++ version 7.0 "Run-Time Library Reference."

To trap floating-point (FP) exceptions via try-except (such as `EXCEPTION_FLT_OVERFLOW`), insert the following before doing FP operations:

```
// Get the default control word.
int cw = _controlfp(0,0);
// Set the exception bits ON.
cw &=~(EM_OVERFLOW|EM_UNDERFLOW|EM_INEXACT|EM_ZERODIVIDE|
EM_DENORMAL);
// Set the control word.
_controlfp(cw, MCW_EM);
```

This turns on all possible FP exceptions. To trap only particular exceptions, choose only the flags that pertain to the exceptions desired.

By default, NT has all the FP exceptions turned off, and thus computations result in NAN or INFINITY rather than an exception. Note, however, that if an exception occurs and an explicit handler does not exist for it, the default exception handler will terminate the process.

If you want to determine which mask bits are set and which are not during exception handling, you need to use `_clearfp()` to clear the floating-point exception. This routine returns the existing FP status word, giving the necessary information about the exception. After this, it is safe to query the chip for the state of its control word with `_controlfp()`. However, as long as an unmasked FP exception is active, most FP instructions will fault, including the `fstcw` in `_controlfp()`. In summary, any handler for FP errors should have `_clearfp()` as its first FP instruction.

Additional reference words: 7.00 3.10 3.1

INF: FormatMessage() Converts GetLastError() Codes

Article ID: Q94999

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The FormatMessage() application programming interface (API) allows you to convert error codes returned by GetLastError() into error strings, using FORMAT_MESSAGE_FROM_SYSTEM in the dwFlags parameter.

Example

// This code sample demonstrates how to get the system message string:

```
LPVOID lpMessageBuffer;
```

```
FormatMessage(  
    FORMAT_MESSAGE_ALLOCATE_BUFFER |  
    FORMAT_MESSAGE_FROM_SYSTEM,  
    NULL,  
    GetLastError(),  
    MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),  
    (LPTSTR) &lpMessageBuffer,  
    0,  
    NULL );
```

```
//... now display this string
```

Additional reference words: 3.10 3.1

INF: Validating User Accounts (Impersonation)

Article ID: Q96005

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Some applications need the ability to execute processes in the context of another user. This impersonation restricts (or expands) the permissions of the account in which the application was executed (file access, permission to change system time, permission to shut down the system, and so forth).

For example, an administrator executes a network server program that allows remote users to log on to the system and perform actions, as if they were logged on to the system locally. Because the administrator initiated the server program and is currently logged on, all actions the server program performs will be in the security context of the administrator. If a guest user logs on remotely, he/she will have all the permissions the administrator account has.

With the Win32 APIs (application programming interfaces), impersonating a remote client is possible only via the `ImpersonateDDEClientWindow()`, `ImpersonateNamedPipeClient()` and `RpcImpersonateClient()` APIs.

More Information:

A common application of impersonation is network server programs (daemons). For example, a remote login daemon needs a user to be able to log in to a remote host and have the host impose all restrictions of the client login account.

If the daemon is using named pipes, dynamic data exchange (DDE), or a remote procedure call (RPC) (using the named pipes transport), the client account may be impersonated on the server daemon, which will impose all the restrictions of the client's user account.

Using other network interfaces (such as Windows sockets--network programming interfaces), security cannot be monitored by the system. A workaround would be to impose password-level security on "login" to the application. The passwords would be maintained by the application in a private accounts database. However, none of the user actions are performed in the security context of the actual client user's account. Therefore, after the server/daemon has validated the client, the server must be careful to only perform actions on behalf of the client that the server knows the client should be allowed to do.

Another option is to create a network share with restricted access. The `WNetAddConnection2()` API can verify access to this system resources [disk or printer network resource (share)]. If the network

share was set up to allow access by a restricted group of people, the WNetAddConnection2() could validate actual user accounts, maintained by Windows NT. As with the previous option, the daemon must be careful to perform only restricted actions on behalf of the client. This option could be used for file server daemons.

Additional reference words: 3.10

INF: Types of File I/O Under Win32

Article ID: Q99173

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.10
-

Summary:

There are multiple types of file handles that can be opened under Win32 and the C run time:

Returned Type	File Creation API	API Set
HANDLE	CreateFile()	Win32
HFILE	OpenFile()/_lcreat()	Win32
int	_creat()/_open()	C run time
FILE *	fopen()	C run time

In general, these file I/O "families" are incompatible with each other. On some implementations of the Win32 application programming interfaces (APIs), the OpenFile()/_lcreat() family of file I/O APIs are implemented as "wrappers" around the CreateFile() family of file I/O APIs, meaning that OpenFile(), _lcreat(), and _lopen() end up calling CreateFile(), returning the handle returned by CreateFile(), and do not maintain any state information about the file themselves. However, this is an implementation detail only and is NOT a design feature.

Note that you cannot count on this being true on other implementations of the Win32 APIs. Win32 file I/O APIs may be written using different methods on other platforms, so reliance on this implementation detail may cause your application to fail.

The rule to follow is to use one family of file I/O APIs and stick with them--do not open a file with _lopen() and read from it with ReadFile(), for instance. This kind of incorrect use of the file I/O APIs can easily be caught by the compiler, because the file types (HFILE and HANDLE respectively) are incompatible with each other and the compiler will warn you (at warning level /w3 or higher) when you have incorrectly passed one type of file handle to a file I/O API that is expecting another, such as passing an HFILE type to ReadFile(HANDLE, ...) in the above example.

More Information:

Compatibility

The OpenFile() family of file I/O functions is provided only for compatibility with earlier versions of Windows. New Win32 applications should use the CreateFile() family of file I/O APIs, which provide added functionality that the earlier file I/O APIs do not provide.

Each of the two families of C run-time file I/O APIs are incompatible with any of the other file I/O families. It is incorrect to open a file handle with one of the C run-time file I/O APIs and operate on that file handle with any other family of file I/O APIs, nor can a C run-time file I/O family operate on file handles opened by any other file I/O family.

`_get_osfhandle()`

For the C run-time unbuffered I/O family of APIs [`_open()`, and so forth], it is possible to extract the operating system handle that is associated with that C run-time handle via the `_get_osfhandle()` C run-time API. The operating system handle is the handle stored in a C run-time internal structure associated with that C run-time file handle. This operating system handle is the handle that is returned from an operating system call made by the C run time to open a file [`CreateFile()` in this case] when you call one of the C run-time unbuffered I/O APIs [`_open()`, `_creat()`, `_sopen()`, and so forth].

The `_get_osfhandle()` C run-time call is provided for informational purposes only. Problems may occur if you read or write to the file using the operating system handle returned from `_get_osfhandle()`; for these reasons we recommend that you do not use the returned handle to read or write to the file.

It is also possible to construct a C run-time unbuffered file I/O handle from an operating system handle [a `CreateFile()` handle] with the `_open_osfhandle()` C run-time API. In this case, the C run time uses the existing operating system handle that you pass in rather than opening the file itself. It is possible to use the original operating system handle to read or write to the file, but it is very important that you use only the original handle or the returned C run-time handle to access the file, but not both, because the C run time maintains state information that will not be updated if you use the operating system handle to read or write to the file.

Additional reference words: 3.10

INF: FILE_READ_EA and FILE_WRITE_EA Specific Types
Article ID: Q102104

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The FILE_READ_EA and FILE_WRITE_EA specific types provide access to read and write a file's extended attributes. Specific access types are represented as bits in the access mask and are specific to the object type associated with the mask.

Please note that these specific types are used in the definition of constants such as FILE_GENERIC_READ, and are not intended to be generally used when specifying access (generic access types are much more appropriate).

Additional reference words: 3.10

INF: Chaining Parent PSP Environment Variables

Article ID: Q96209

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Some MS-DOS applications change the environment variables of their parent application by chaining through the program segment prefix (PSP). With Windows NT, this functionality doesn't work if the parent is a 32-bit application.

When an MS-DOS application is started from a single command shell (SCS), the application inherits a new copy of the environment variables. Any attempts by the MS-DOS application to modify its parent's environment variables will not work. When the MS-DOS application exits, the SCS will be "restored" to its original state. If another MS-DOS application is started, the second application will receive the same environment that the first MS-DOS application received.

If an MS-DOS application (B) is spawned by another MS-DOS application (A), any modifications to application A's environment variables will be reflected when application B exits.

More Information:

For more information on how environment variables are set, query on the following words in the Microsoft Knowledge Base:

set and environment and variables and Windows and NT

Additional reference words: 3.10

INF: System GENERIC_MAPPING Structures

Article ID: Q102105

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

There is not a Win32 application programming interface (API) to retrieve the GENERIC_MAPPING structures for Windows NT objects. The MapGenericMask() Win32 API is intended to use GENERIC_MAPPING structures associated with private objects created by the application.

Additional reference words: 3.10

INF: Default Stack in Win32 Applications

Article ID: Q97786

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

By default, space is reserved for applications in the following manner:

- 1 megabyte (MB) reserved (total virtual address space for the stack)
- 4K committed (total physical memory allocated when stack is created)

Note: The `-heap` linker option can be used to modify both of these values.

The operating system will grow the stack as needed by committing 4K blocks out of the reserved stack memory. Once all of the reserved memory has been committed, Windows NT will attempt to continue to grow the stack into the memory adjacent to the memory reserved for the stack, as shown in the following:

```
|<--- Total 1 MB for stack --->|<--- Adjacent memory --->|
-----
| 4K | 1020K ... | ... |
|    |         |   |   |
-----
```

However, once the stack grows to the point that the adjacent area is not free (and this may happen as soon as the reserved 1 MB has been committed), the stack cannot grow any farther. Therefore, it is very risky to rely on this memory being free. Applications should take care to reserve all the memory that will be needed by increasing the amount of memory reserved for the stack.

In other cases, it may be desirable to reduce the amount of memory reserved for the stack.

The `/STACK` option in the linker and the `STACKSIZE` statement in the DEF file can be used to change both the amount of reserved memory and the amount of committed memory. The syntax for each method is shown below:

```
/STACK:[reserve][,commit]
```

```
STACKSIZE [reserve][,commit]
```

More Information:

Each new thread gets its own stack space of committed and reserved memory. If a new size is not specified in the `CreateThread()` call, the new thread takes on the same stack size as the thread that created it, whether that be the default value, a value defined in the DEF file, or by the linker switch.

The system handles committing more reserved stack space when needed, but cannot reserve or commit more than the total amount initially reserved (or committed if no additional is reserved). Remember that the only resource consumed by reserving space is addresses in your process. No memory or pagefile space is allocated. When the memory is actually committed, both memory and pagefile resources are allocated. There is no harm in reserving a large area if it might be needed.

As always, automatic variables are placed on the stack. All other static data is located in the process address space. Because they are static, they do not need to be managed like heap memory.

Note that under Win32s, stacks are limited to a maximum of 128K.

Additional reference words: 3.10

PRB: Code in DLL Causes Access Violation C0000005

Article ID: Q97787

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

A DLL entry point with `__cdecl` convention results in an access violation (specifically, C0000005).

CAUSE

Windows NT expects that the dynamic-link library (DLL) entry point will have a calling convention of `__stdcall`. If the entry point is not explicitly given a calling convention, the compiler's default convention is `__cdecl`.

This is not an obvious mistake to track down. Under the debugger, you will notice that you can trace into the entry point, but the next function call will fail. Thus, it appears as if the execution of the entry point succeeded. The problem is really that the stack was not properly cleaned up.

RESOLUTION

To correct the problem, do one of the following

- Add `WINAPI` to the function prototype and declaration so that the function will have the `__stdcall` calling convention.

-or-

- Compile with `-Gz` so that by default, all functions will be declared with the `__stdcall` convention. If you are using `NTWIN32.MAK`, the macro `"scall"` is defined for this purpose.

In addition, the entry point should be specified with the `-entry` linker switch. For example, if the entry point is named `LibMain()`, add the following to the link command line:

```
-entry:LibMain$(DLENTY)
```

More Information:

The same problem occurs if callback functions, such as dialog box procedures, are not declared `__stdcall`. The solutions described above apply in this situation as well.

Additional reference words: 3.10

INF: Starting and Terminating 16-Bit Windows Applications
Article ID: Q105676

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

A 16-bit Windows application running under Windows NT is running as a thread in a single virtual MS-DOS machine (VDM). These threads are nonpreemptively scheduled. A 16-bit Windows application shares an address space and an input queue with other 16-bit Windows applications. Objects created by a thread (application) are owned by the thread (application). This environment is called WOW (Windows on Win32).

When a 16-bit application is started via `CreateProcess()`, the process handle and the thread handle contained in the `PROCESS_INFORMATION` structure are pseudo-handles. The only application programming interfaces (APIs) that can use the process handle are `WaitForSingleObject()`, `WaitForMultipleObjects()`, and `WaitForInputIdle()`.

A common question is "How can I terminate a 16-bit process from a 32-bit process?" However, as implied above, `PROCESS_INFORMATION.hProcess` cannot be used in `TerminateProcess()` and `PROCESS_INFORMATION.dwThreadId` cannot be used in `PostThreadMessage()`.

One way to terminate an individual 16-bit Windows application is to enumerate the desktop windows using `EnumWindows()`, determine which is the correct window, obtain the thread ID with `GetWindowThreadProcessId()`, and post a `WM_QUIT` message via `PostThreadMessage()` to terminate the application.

Additional reference words: 3.10

INF: Why LoadLibraryEx() Returns an HINSTANCE

Article ID: Q102128

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

In the Win32 Help files, LoadLibrary() is typed to return a HANDLE, while LoadLibraryEx() is prototyped to return a HINSTANCE.

An HINSTANCE return from LoadLibraryEx() is useful because processes that load dynamic-link libraries (DLLs) do not necessarily want the overhead of having to page in code for a DllEntryPoint routine when the DLL does not need to initialize information. This is especially useful when you have multiple threads that attach to already loaded DLLs. In this case, you may want to not implicitly load via LoadLibrary() and instead use LoadLibraryEx() to explicitly load without having to page in the code for every attach.

LoadLibraryEx() is also useful if you want to retrieve resources from a DLL or an EXE. In this case, you would use LoadLibraryEx() to load the module you want into your address space, without executing DllEntryPoint, and then use the resource application programming interfaces (APIs) to access the data.

Additional reference words: 3.10

INF: CTRL+C Exception Handling Under WinDbg

Article ID: Q97858

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

An exception is raised for CTRL+C only if the process is being debugged. The purpose is to make it convenient for the debugger to catch CTRL+C in console applications. For the purposes of this article, the debugger is assumed to be WinDbg.

When the console server detects a CTRL+C, it uses `CreateRemoteThread()` to create a thread in the client process to handle the event. This new thread then raises an exception IF AND ONLY IF the process is being debugged. At this point, the debugger either handles the exception or it continues the exception unhandled.

The "gh" command marks the exception as having been handled and continues the execution. The application does not notice the CTRL+C, with one exception: CTRL+C causes alertable waits to terminate. This is most noticeable when executing:

```
while( (c = getchar()) != EOF ) - or - while( gets(s) )
```

It is not possible to get the debugger to stop the wait from terminating.

The "gn" command marks an exception as unhandled and continues the execution. The handler list for the application is searched, as documented for `SetConsoleCtrlHandler()`. The handler is executed in the thread created by the console server.

After the exception is handled, the thread created to handle the event terminates. The debugger will not continue to execute the application if Go On Thread Termination is not enabled (from the Options menu, choose Debug, and select the Go On Thread Termination check box). The thread and process status indicate that the application is stopped at a debug event. As soon as the debugger is given a go command, the dead thread disappears and the application continues execution.

More Information:

There are three cases where CTRL+C doesn't cause the program to stop executing (instead it causes a "page down"):

1. When CTRL+C is already being handled.
2. When the debugger is in the foreground and a source window has the focus (both must be true).
3. When the CTRL+C exception is disabled (through the Debugger

Exceptions dialog box).

This follows the convention of the WordStar/Turbo C/Turbo Pascal editor commands.

Additional reference words: 3.10

PRB: New Parameter for the CreateService() API

Article ID: Q97921

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The online Win32 application programming interface (API) reference provided with the Win32 Software Development Kit (SDK) March 1993 preliminary release incorrectly documents CreateService() as having 12 parameters. The header file WINSVC.H correctly documents CreateService() as having a 3rd parameter named lpDisplayName.

The lpDisplayName parameter is the string that is displayed by the Service application and when you use the command line "net start|stop|pause|continue" syntax.

Additional reference words: 3.10

INF: The Use of the SetLastErrorEx() API

Article ID: Q97926

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SetLastErrorEx() is intended for better debugging support, not for passing additional error information. This function is not fully realized under the current version of Windows NT.

The SetLastErrorEx() application programming interface (API) differs from the SetLastError() API only in that it raises a debug "RIP" event. The RIP event is intended to give text to the debugger so that the user can retry, ignore, and so forth, these errors. SetLastErrorEx() raises an exception only if SetDebugErrorLevel() has been called by the debugger to allow the errors to be passed on.

The error type can be determined from the debugger by examining the debug event structure that is passed with the event. The debug event structure contains a RIP_INFO substructure.

Additional reference words: 3.10

INF: Passing a Pointer to a Member Function to the Win32 API
Article ID: Q102352

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Many of the Win32 application programming interfaces (APIs) call for a callback routine. One example is the lpStartAddr argument of CreateThread():

```
HANDLE CreateThread(lpsa, cbStack, lpStartAddr, lpvThreadParm,
                   fdwCreate, lpIDThread)

LPSECURITY_ATTRIBUTES lpsa;    /* Address of thread security attrs */
DWORD cbStack;                /* Initial thread stack size*/
LPTHREAD_START_ROUTINE lpStartAddr; /* Address of thread function */
LPVOID lpvThreadParm;         /* Argument for new thread*/
DWORD fdwCreate;              /* Creation flags*/
LPDWORD lpIDThread;          /* Address of returned thread ID */
```

When attempting to use a member function as the thread function, the following error is generated:

```
error C2643: illegal cast from pointer to member
```

The problem is that the function expects a C-style callback, not a pointer to a member function. A major difference is that member functions are called with a hidden argument called the "this" pointer. In addition, the format of the pointer isn't simply the address of the first machine instruction, as a C pointer is. This is particularly true for virtual functions.

If you want to use a member function as a callback, you can use a static member function. Static member functions do not receive the "this" pointer and their addresses correspond to an instruction to execute.

Static member functions can only access static data, and therefore to access nonstatic class members, the function needs an object or a pointer to an object. One solution is to pass in the "this" pointer as an argument to the member function.

MORE INFORMATION

=====

This situation occurs with callback functions of other types as well, such as:

DLGPROC

GRAYSTRINGPROC

EDITWORDBREAKPROC	LINEDDAPROC
ENHMFENUMPROC	MFENUMPROC
ENUMRESLANGPROC	PROPENUMPROC
ENUMRESNAMEPROC	PROPENUMPROCEX
ENUMRESTYPEPROC	TIMERPROC
FONTENUMPROC	WNDENUMPROC
GOBJENUMPROC	

For more information on C++ callbacks, please see the May issue of the "Windows Tech Journal."

The following sample demonstrates how to use a static member function as a thread function, and pass in the "this" pointer as an argument.

Sample Code

```

-----
#include <windows.h>

class A
{
public:
    int x;
    int y;

    A() { x = 0; y = 0; }

    static StartRoutine( A * );    // Compiles clean, includes "this" pointer
};

void main( )
{
    A a;

    DWORD dwThreadID;

    CreateThread( NULL,
        0,
        (LPTHREAD_START_ROUTINE) (a.StartRoutine),
        &a,
        // Pass "this" pointer to static member fn
        0,
        &dwThreadID
    );
}

```

Additional reference words: 3.10

INF: File Manager Passes Short Filename as Parameter

Article ID: Q98575

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When starting an application from File Manager by double-clicking a document associated with the application, if the document resides on an NTFS partition and has a long (non-8.3 form) filename, File Manager will pass the short version of the filename (also known as the MS-DOS alias or 8.3 name) to the associated application. This is done for compatibility reasons; applications not aware of long filenames (16-bit applications) can still function correctly.

This can create confusion, however, if the application displays the name of the file the application was started with; the short name is displayed even though the long name was double-clicked.

You can avoid possible confusion by always expanding any filenames passed to an application via the command line. Do this by calling the FindFirstFile() application programming interface (API) on these filenames. FindFirstFile() will always return the file system's version of the filename in the WIN32_FIND_DATA.cFileName structure member, which the application can then use in all further references to the file without any problems.

Additional reference words: 3.10 file name

INF: Windows NT Virtual Memory Manager Uses FIFO

Article ID: Q98216

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

On page 193 of "Inside NT," Helen Custer states that the Windows NT virtual memory manager uses a FIFO (first in, first out) algorithm, as opposed to a LRU (least recently used) algorithm, which the Windows virtual memory manager uses. While it is true that FIFO can result in a commonly used page being discarded or paged to the pagefile, there are reasons why this algorithm is preferable.

Here are some of the advantages:

- FIFO is done on a per-process basis; so at worst, a process that causes a lot of page faults will slow only itself down, not the entire system.
- LRU creates significant overhead--the system must update its page database every single time a page is touched. However, the database may not be properly updated in certain circumstances. For example, suppose that a program has good locality of reference and uses a page constantly so that it is always in memory. The operating system will not keep updating the timestamp in the page database, because the process is not hitting the page table. Therefore this page may age even though it is in nearly constant use.
- Pages that are "discarded" are actually kept in memory for a while, so if a page is really used frequently, it will be brought back into memory before it is written to disk.

Additional reference words: 3.10 file

INF: Determining Memory Usage Under Windows NT

Article ID: Q98721

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following are tools available in Windows NT that can help you determine memory usage in your system:

PMON - Allows you look at all the processes on the system, and monitor their memory usage and private commitment. This tool is character-mode-based and updates every 5 seconds.

PVIEW - Provides information at an individual process's address space and memory usage. This function provides more detail than PMON.

PWALK - A GUI tool allowing you to examine the address space of your application or another running .EXE. See the Process Walker icon in the Win32 SDK group in Program Manager. Process Walker provides more detail than PVIEW.

More Information:

The easiest way for you to determine your memory usage is to run PMON. Here's a sample output screen from PMON:

```
Process Monitor: Total Memory: 16192Kb Available 4348Kb PgFlts:38  
Commit: 31692Kb / 19500Kb Limit: 40028Kb Pool: 1508Kb /2044Kb
```

%CPU	CPU Time	Mem Usage	Mem Diff	Page Faults	Flts Diff	Commit Charge	Thd Cnt	Image Name
		896	0	2201408	0			File Cache
82	10:21:10	16	0	0	0	0	1	Idle Process
0	0:23:52	16	0	771	0	32	20	System Process
0	0:00:00	0	0	106	0	192	6	smss.exe
6	0:54:15	2140	0	55528	0	3832	29	csrss.exe
0	0:00:01	0	0	550	0	456	2	winlogon.exe
0	0:00:06	0	0	2044	0	952	4	screg.exe
0	0:00:07	0	0	8153	0	592	8	lsass.exe
0	0:00:25	0	0	6185	0	640	4	spoolss.exe
0	0:00:00	0	0	1193	0	348	5	EventLog.exe
0	0:00:00	0	0	245	0	332	2	mcsxnsv.exe
0	0:00:08	0	0	8830	0	264	2	ubnbsvc.exe
0	0:00:00	0	0	713	0	552	7	netdde.exe
0	0:00:01	16	0	7	0	4	8	No Name Found
0	0:00:00	0	0	495	0	316	2	clipsrv.exe
0	0:00:20	120	0	16892	0	652	12	lmsvcs.exe
0	0:00:00	0	0	763	0	484	7	MsgSvc.exe
0	0:00:00	0	0	450	0	296	1	nddeagnt.exe
0	0:00:01	0	0	926	0	240	1	taskman.exe

0	0:00:03	348	0	1805	0	324	2	progman.exe
0	0:00:31	0	0	5784	0	224	1	CMD.exe
0	0:00:50	156	0	13712	0	120	1	PERFMTR.EXE
0	0:00:03	0	0	912	0	144	1	CMD.exe
0	0:01:15	20	0	5379	0	4052	4	ntvdm.exe
0	0:00:03	0	0	1352	0	156	1	CMD.exe
0	0:15:28	308	0	43509	0	280	1	PMON.EXE
0	0:00:47	252	0	16776	0	1828	2	I386KD.EXE
2	0:05:17	2652	0	3056	0	900	3	MSMAIL32.EXE
0	0:01:07	224	0	23298	0	600	3	MAILSP32.EXE
0	0:00:00	20	0	199	0	328	1	STATUS.EXE
7	0:00:14	1060	0	1529	38	360	1	slm.exe

The first line indicates that the current machine has 16 megabytes (MB) of memory, of which 4348K is unused, and there have been 38 page faults since the last update.

The second line indicates that the total commitment is 31692K with the private commitment at 19500K. Private commits are writable pages that are not shared; they are private to that process, usually created with LocalAlloc() or VirtualAlloc(). The difference between these two numbers, in this case 12196K, is the amount of memory used by pagable drivers, paged pool, and page-file-backed shared memory regions. Drivers have about 1 MB of pagable memory, and the system we are examining has about 2 MB of paged pool; therefore, subtracting 3 MB from the 12 MB of other commitment, yields 9 MB of shared virtual memory.

The total commit limit is 40028K. Nonpaged pool is using 1508K; paged pool is using 2044K.

The rest of the screen describes the memory consumption on a process-by-process basis. The Mem Usage column is the amount of physical memory in kilobytes that application is using. If it is 0, the application has been entirely paged out of memory. The Mem Diff column is the difference in memory usage in kilobytes over the last 5 seconds. The other column of interest is the Commit Charge. This is the total number of writable nonshared pages in the process.

From a quick scan of the list, the Windows subsystem CSRSS.EXE is consuming just over 2 MB with almost 4 MB committed. MSMAIL32 is consuming around 2.5 MB with 3 MB committed. NTVDM is consuming 20K with 4 MB committed.

The larger these numbers are, the poorer the system performance. CSRSS generally is between 2 and 3 MB with 3 to 4 MB committed. The more memory you have, the more memory an application is given. Therefore, if you have 2 MB, CSRSS will consume about 5 MB. If your system seems to be sluggish and PMON shows numbers that seem excessive, you should check for memory leaks. Memory leaks happen when certain commands are executing over and over causing huge commitments in certain processes.

Additional reference words: 3.10

INF: Getting the Net Time on a Domain

Article ID: Q98722

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When trying to do a

```
net time /domain:egdomain /set
```

you may get a message saying the account is not known or the password is invalid. This can happen if you are logged on using an account whose name is spelled "Administrator", but the account is a different Administrator account than the one on the domain controller. For example, if you are logged on as EGMACHINE\Administrator, and attempt

```
net time /domain:egdomain /set
```

you will get an error message because EGMACHINE\Administrator is not the same account as EGDOMAIN\Administrator.

The solution is to log off of EGMACHINE, log back on as EGMACHINE\PowerUsr1, then execute the command. Note that a privilege is needed to set the time on a machine. In the previous example, the account, EGMACHINE\PowerUsr1, was used to remind us that power users have the needed privilege.

More Information:

When running Windows NT while logged on to a domain, doing a NET TIME without the /DOMAIN parameter, as mentioned above, probably will not yield the desired results. However, because you are logged on to a domain, you can do

```
net time /domain /set
```

and a domain controller from the domain you are logged on to will be used. In other words, if you are logged on to a domain, the /DOMAIN parameter is necessary, but the actual domain name can optionally be left to default to the domain you're currently participating in. If your machine is joined to the a domain, that domain will be the default domain for NET TIME /DOMAIN.

Additional reference words: 3.10

INF: Noncontinuable Exceptions

Article ID: Q98840

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

An exception is noncontinuable if the event isn't continuable in the hardware, or if continuation makes no sense. For example, if the caller's stack is corrupted while trying to post an exception, continuing from the bad stack exception would not be useful.

The noncontinuable exception does not terminate the application, and therefore an application that can succeed in catching the exception and running after a noncontinuable exception is free to do so. However, a noncontinuable exception typically arises as a result of a corrupted stack or other serious problem, making it very difficult to recover from the exception.

Additional reference words: 3.10 non-continuable

INF: Validating User Account Passwords Under Windows NT
Article ID: Q98891

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Windows NT stores account names and passwords in the security accounts manager (SAM) database. Windows NT checks this database to validate passwords when users log on.

At this time, there is no nonprivileged service that takes a user name and a password and returns an indication of whether or not the user account password is valid. There is a privileged service that handles this password validation; it is for use by logon processes such as winlogon. This service is not yet published. A nonprivileged service that will perform password validation is under consideration for the versions of Windows NT that will follow the first released product.

More Information:

The SAM application programming interface (API) functions were not exposed due to their changing nature. Microsoft is working on a developer's kit that will provide guidelines and tutorial information about most of the security API functions, including the SAM APIs.

Exposing the SAM API will not compromise security because the passwords are encrypted within SAM; they are one-way encrypted such that not even SAM can decrypt them. Even a dictionary attack (encrypt an entire dictionary and see if any of the words match) would not be easy, because there is no SAM API function that will read the encrypted password.

Additional reference words: 3.10 3.1 non-privileged

PRB: Unexpected Result of SetFilePointer() with Devices
Article ID: Q98892

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

Open a floppy drive with CreateFile():

```
h = CreateFile( "\\.\a:", ... );
```

Use SetFilePointer() to advance the file pointer associated with the file handle returned from CreateFile():

```
SetFilePointer( h,      // file handle
                5,      // distance (in bytes) to move file pointer
                NULL,    // optional high 32-bits of distance
                FILE_BEGIN // specifies the starting point
            );
```

If the offset is not a multiple of the sector size of the floppy drive, the function will return success; however, the pointer will not be exactly where requested. The pointer value is rounded down to the beginning of the sector that the pointer value is in.

CAUSE

The behavior of this application programming interface (API) is by design for the following reasons:

- The I/O system is unaware of device particulars such as sector size; any offset is valid.
- SetFilePointer() is very frequently used. Because speed is an important goal for Windows NT, time is not spent on querying device particulars and detecting such errors.
- The logic to handle this situation is built into the file system, which actually performs the rounding, and therefore there was no need to put this into the code for SetFilePointer().

RESOLUTION

When using SetFilePointer() with a handle that represents a floppy drive, the offset must be a multiple of the sector size for the floppy drive in order for the function to perform as expected.

More Information:

Think of a file pointer as merely a stored value, which is where the next read or write will take place. In fact, it is possible to

override this value on either the read or write itself, using certain APIs, by supplying a different location. The new pointer location is remembered after the operation. Therefore, the operation of "setting a file pointer" merely means to go store a large integer in a cell in the system's data structures, for possible use in the next file operation. In the case of a handle to a device, the file pointer must be on a sector boundary.

In a similar way, ReadFile() only reads amounts that are multiples of the sector size if it is passed a handle that represents a floppy drive.

Additional reference words: 3.10 3.1

INF: Limit on the Number of Bytes Written Asynchronously
Article ID: Q98893

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

There is a limit to the number of bytes that can be written with WriteFile() using asynchronous I/O (FILE_FLAG_OVERLAPPED specified). This limit depends on the size of your system.

Asynchronous (overlapped) I/O consumes system resources for a long time. For example, the memory used is locked in the process working set until the I/O completes. To limit the amount of system resources used asynchronously by an application, the system charges asynchronous I/O to the working set of the process requesting the I/O.

While the working set size is dynamically raised and lowered based on the load, there are minimum and maximum values. These values are based on system size: consider up to 12 megabytes (MB) a small system, between 12 MB and 19 MB a medium system, and greater than 19 MB a large system. Each process is guaranteed a minimum working set for performance reasons; about 120K for small systems, 160K for large systems, and 245K for large systems.

When system resources are heavily taxed, a process is confined to its maximum working set. Asynchronous I/Os may never cause you to exceed your maximum working set, because once you are allowed to initiate an asynchronous I/O, the page cannot be taken away if memory becomes tight. The maximum working set sizes are about 300K for a small system, 716K for a medium system, and 1.5 MB for a large system.

More Information:

The following code can be used to experiment with the maximum number of bytes that can be written using asynchronous I/O. Simply change the line to vary the number of bytes that the code attempts to write:

```
#define NBR_BYTE 700000
```

Sample Code

```
#include <stdio.h>  
#include <stdlib.h>  
#include <malloc.h>  
#include <assert.h>
```

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

```
#define NBR_BYTE 700000
```

```

int main(void)
{
    char          *c;
    HANDLE        hFile;
    DWORD        byteWrite;
    OVERLAPPED    overLap;
    DWORD        err;
    BOOL         result;

    c = malloc( NBR_BYTE );
    assert( c != NULL );

    overLap.hEvent = CreateEvent( NULL, FALSE, FALSE, "event1" );
    assert( overLap.hEvent );

    hFile = CreateFile( "test", GENERIC_WRITE, 0, NULL, OPEN_ALWAYS,
                       FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED |
                       FILE_FLAG_WRITE_THROUGH,
                       NULL);

    if( hFile == INVALID_HANDLE_VALUE )
    {
        free( c );
        printf( "error opening file\n" );
        exit( 0 );
    }

    overLap.Offset      = 0;
    overLap.OffsetHigh = 0;
    result = WriteFile( hFile, c, NBR_BYTE, &byteWrite, &overLap );
    if( result == FALSE )
    {
        err = GetLastError( );
        if( err != ERROR_IO_PENDING )
        {
            free( c );
            printf( "Error: %d\n", GetLastError() );
            exit( 0 );
        }
    }

    free( c );

    return 0;
}

```

Additional reference words: 3.10 3.1 asynch

INF: Setting File Permissions

Article ID: Q98952

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In Windows NT, local access controls can be set on just NTFS partitions, not FAT or HPFS partitions or floppies. Read/execute-only permissions should work properly on a CD-ROM.

The exception is that ACLs (access control lists) can be set on shares, regardless of the file system, to control access to all the files within that share. For example, you can give read access to everyone, but give full access just to members of a certain group or to certain individuals.

Additional reference words: 3.10 3.1

INF: Detecting Closure of Command Window from a Console App
Article ID: Q102429

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

Win32 console applications run in a command window. For the console application to detect when the console is closing, register a console control handler and look for the following values in your case statement:

CTRL_CLOSE_EVENT	User closes the console
CTRL_LOGOFF_EVENT	User logs off
CTRL_SHUTDOWN_EVENT	User shuts down the system

For an example, see the CONSOLE sample. For more information, see the entry for SetConsoleCtrlHandler() in the Win32 application programming interface (API) reference.

Additional reference words: 3.10

INF: Definition of a Protected Server

Article ID: Q102447

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The Win32 application programming interface (API) reference briefly discusses creating a "protected server" that assigns security to private objects. This article explains the concept of a protected server" and its relationship to private objects.

A protected server is an application that provides services to clients. These services could be as simple as saving and retrieving information from a database while issuing security checks to verify that the client has proper access.

A private object is an application-defined data structure that both the client and server recognize. Private objects are not registered with nor recognized by the Windows NT operating system; they are entirely application-defined.

It is not uncommon for security to be assigned to private objects in a protected server's database. For example, when a client asks the server to create a new object in the database, the server could use the `CreatePrivateObjectSecurity()` Win32 API to create a security descriptor (SD) for the new private object. The server would then store the SD with the private object in the database. It is important to note that there is nothing in the SD that associates it with the private object. Instead, it is up to the protected server to maintain that association in the private object or in the database. It is likely that the private object and the associated SD would be stored together in a single database record.

A protected server application is responsible for checking a client's access before providing information. For example, when a client asks the server to retrieve some data, the server would go out and locate the record (which would contain the private object and SD) and bring a copy of the SD into memory. It would then call the `AccessCheck()` Win32 API passing the SD, the client's access token, and the desired access mask. `AccessCheck()` will check the client's access against the object's SD to determine if access is permitted. Depending on the result of `AccessCheck()`, the protected server would either provide the requested information or deny access.

In conclusion, a protected server is an application that performs operations on private objects that are entirely user defined. The protected server is responsible for associating security descriptors to those objects and must take the steps necessary to verify a client's access.

Additional reference words: 3.10

INF: SetTimer() Should Not Be Used in Console Applications
Article ID: Q102482

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

SetTimer() was not designed to be used with a console application because it requires a message loop to dispatch the timer signal to the timer procedure. In a console application, this behavior can be easily emulated with a thread that is set to wait on an event.

SetTimer() can work within a console application, but it requires a thread in a loop calling GetMessage() and DispatchMessage().

For example,

```
while (1)
{
    GetMessage();
    DispatchMessage();
}
```

Because this requires a thread looping, there is no real advantage to adding a timer to a console application over using a thread waiting on an event.

Another option is to use a multimedia timer, which does not require a message loop and has a higher resolution. See the help for timeSetEvent() and the Multimedia overview. Any application using Multimedia calls must include MMSYSTEM.H, and must link with WINMM.LIB.

Additional reference words: 3.10

INF: Security Attributes on Named Pipes

Article ID: Q102798

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The March release of the Windows NT beta (and earlier) does not require security attributes on pipes. It was valid at that time to enter NULL for the last parameter of the Win32 application programming interface (API) CreateNamedPipe().

Windows NT now requires security attributes for pipes. Please note that setting the security attributes parameter to NULL does not indicate that you want a NULL security descriptor (SD), rather it indicates that you want to inherit the security descriptor of the current access token. This generally means that any client wanting to connect to your pipe server must have the same security attributes as the user that started the server. For example, if the user who started the server was the administrator of the machine, then any client who wants to connect must also be an administrator to that machine.

Below is an code sample that demonstrates creating a named pipe with a NULL security descriptor.

```
HANDLE                hPipe;    // Pipe handle.
SECURITY_ATTRIBUTES  sa;        // Security attributes.
PSECURITY_DESCRIPTOR pSD;      // Pointer to SD.

// Allocate memory for the security descriptor.

pSD = (PSECURITY_DESCRIPTOR) LocalAlloc(LPTR,
                                        SECURITY_DESCRIPTOR_MIN_LENGTH);

// Initialize the new security descriptor.

InitializeSecurityDescriptor(pSD, SECURITY_DESCRIPTOR_REVISION);

// Add a NULL descriptor ACL to the security descriptor.

SetSecurityDescriptorDacl(pSD, TRUE, (PACL) NULL, FALSE);

sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = pSD;
sa.bInheritHandle = TRUE;

// Create a local named pipe with a NULL security descriptor.

hPipe = CreateNamedPipe(
    "\\.\PIPE\test",    // Pipe name = 'test'.
```

```
PIPE_ACCESS_DUPLEX      // 2-way pipe.
| FILE_FLAG_OVERLAPPED, // Use overlapped structure.
PIPE_WAIT               // Wait on messages.
| PIPE_READMODE_MESSAGE // Specify message mode pipe.
| PIPE_TYPE_MESSAGE,
MAX_PIPE_INSTANCES,    // Maximum instance limit.
OUT_BUF_SIZE,          // Buffer sizes.
IN_BUF_SIZE,
TIME_OUT,              // Specify time out.
&sa);                 // Security attributes.
```

It is important to note that by specifying TRUE for the fDaclPresent parameter and NULL for pAcl parameter of the SetSecurityDescriptorDacl() API, a NULL access control list (ACL) is being explicitly specified.

Additional reference words: 3.10

INF: Using Temporary File Can Improve Application Performance
Article ID: Q103237

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The use of temporary files can significantly increase the performance of an application. By using `CreateFile()` with the `FILE_ATTRIBUTE_TEMPORARY` flag, you let the system know that the file is likely to be short lived. The temporary file is created as a normal file. The system needs to do a minimal amount of lazy writes to the file system to keep the disk structures (directories and so forth) consistent. This gives the appearance that the file has been written to the disk. However, unless the Memory Manager detects an inadequate supply of free pages and starts writing modified pages to the disk, the Cache Manager's Lazy Writer may never write the data pages of this file to the disk. If the system has enough memory, the pages may remain in memory for any arbitrary amount of time. Because temporary files are generally short lived, there is a good chance the system will never write the pages to the disk.

To further increase performance, your application might mark the file as `FILE_FLAG_DELETE_ON_CLOSE`. This indicates to the system that when the last handle of the file is closed, it will be deleted. Although the system generally purges the cache to ensure that a file being closed is updated appropriately, because a file marked with this flag won't exist after the close, the system foregoes the cache purge.

Additional reference words: 3.20

INF: Calling a Win32 DLL from a Win16 Application Under WOW
Article ID: Q104009

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

Under Windows NT, it is possible to call routines in a Win32 dynamic-link library (DLL) from a Win16 application using an interface called Windows on Win32 (WOW) Generic Thunking. This is not to be confused with Win32s Universal Thunks, which provides this functionality under Windows 3.1.

WOW presents a few new 16-bit application programming interfaces (APIs) that allow you to load the Win32 DLL, get the address of the DLL routine, call the routine (passing it up to thirty-two 32-bit arguments), convert 16:16 (WOW) addresses to 0:32 addresses (useful if you need to build up a 32-bit structure that contains pointers and pass a pointer to it), and free the Win32 DLL.

The Generic Thunks specification is included on the Win32 SDK CD in \DOC\SDK\MISC\GENTHUNK.TXT. However, this document does not include correct information for all of the function prototypes. The following prototypes should be used:

```
DWORD FAR PASCAL LoadLibraryEx32W( LPCSTR, DWORD, DWORD );
DWORD FAR PASCAL GetProcAddress32W( DWORD, LPCSTR );
DWORD FAR PASCAL CallProc32W( DWORD, LPVOID, DWORD, DWORD );
DWORD FAR PASCAL GetVDMPointer32W( LPVOID, UINT );
BOOL FAR PASCAL FreeLibrary32W( DWORD );
```

Note that although these functions are called in 16-bit code, they need to be provided with 32-bit handles, and they return 32-bit handles.

In addition, be sure that your DLL routines are declared with the `_stdcall` convention; otherwise, you will get an access violation.

NOTE: It is a good idea to test the 32-bit DLL by calling it from a 32-bit application before attempting to call it from a 16-bit application since the debugging support is superior in the 32-bit environment.

MORE INFORMATION

=====

The following code fragments can be used as a basis for Generic Thunks. Assume that the Win16 application is named `app16`, that the Win32 DLL is named `dll32`, and that the following are declared:

```
typedef void (FAR PASCAL *MYPROC) (LPSTR);

DWORD ghLib;
MYPROC hProc;
char FAR *TestString = "Hello there";
```

The DLL routine is defined in dll132.c as follows:

```
void WINAPI myPrint( LPTSTR lpString )
{
    MessageBox( GetFocus(), lpString, "dll132", MB_OK|MB_SYSTEMMODAL );
}
```

Attempt to load the library in the app16 WinMain():

```
if( NULL == (ghLib = LoadLibraryEx32W( "dll132.dll", NULL, 0 )) ) {
    MessageBox( NULL, "Cannot load DLL32", "App16", MB_OK );
    return 0;
}
```

Attempt to get the address of myPrint():

```
if( NULL == (hProc = (MYPROC)GetProcAddress32W( ghLib, "myPrint" )) ) {
    MessageBox( hWnd, "Cannot call DLL function", "App16", MB_OK );
    ...
}
```

Call myPrint() and pass it TestString as an argument:

```
CallProc32W( (DWORD) TestString, hProc, 1, 1 );
```

Free the library right before exiting WinMain():

```
FreeLibrary32W( ghLib );
```

NOTE: When linking the Win16 application, you need to put the following statements in the .DEF file, indicating that the functions will be imported from the WOW kernel:

```
IMPORTS
    kernel.LoadLibraryEx32W
    kernel.FreeLibrary32W
    kernel.GetProcAddress32W
    kernel.GetVDMPointer32W
    kernel.CallProc32W
```

Additional reference words: 3.10

INF: Dynamically Growing Named File Mappings

Article ID: Q104012

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

CreateFileMapping() can be used to create a named file mapping object, which is generally used as shared memory between applications. This call can use/grow an existing file on disk, use a temporary file it first creates, or use part of the system pagefile.

Mapping a view of the file with MapViewOfFile() maps a range of virtual memory addresses to the appropriate file. Anything that is written to the pointer returned from MapViewOfFile() will be written to the file on disk. All applications with views to this file mapping object will see the same data. It is important to remember that you cannot force the view beyond the end of the file mapping object.

If you have a named file mapping object that needs to be bigger, the file can be grown by calling CreateFileMapping() a second time, specifying a larger size than was specified the previous time. Be sure to specify (as a first parameter) both a handle to the same file that was specified originally and the same name. Otherwise, this file will be grown to the size specified, but it will not be used as the file mapping object. Note that the handle does not need to be closed to do this.

The handle that you receive will be a handle to the original file mapping object, although the handle itself will be different.

MORE INFORMATION

=====

Be sure to check the return code to verify that a file mapping object was created. If you have a handle, it is still a good idea to call GetLastError(). If GetLastError() returns 0 (zero) when creating a named object, this indicates that you have been given the single handle to this file mapping object. If it returns ERROR_ALREADY_EXISTS, then you have been given another handle to an existing file mapping object. If this is intentional (perhaps you are trying to grow the mapping as described above) and you specified the correct handle in the first parameter of CreateFileMapping(), then you can use the handle safely. If you did not intend to be using an existing file mapping object, chances are that you will have problems. In this case, release the handle just received by calling CloseHandle(), and call specifying a new name for the file mapping object.

Additional reference words: 3.10

INF: How Keyboard Data Gets Translated

Article ID: Q104316

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Keyboard input is acquired by the keyboard driver, which in turn produces a scan code. This scan code is passed on to the locale-specific Win32 subsystem keyboard driver. This locale-specific driver then converts the scan code to a virtual key and a Unicode character. The Win32 subsystem then passes on this information to the application.

All messages in the Win32 application programming interface (API) that present textual information to a window procedure depend upon how the window registered its class. For instance, if RegisterClassW() was called, then Unicode is presented; if RegisterClassA() was called, then ANSI is presented. The conversion of the text is handled by the Window Manager. This allows an ANSI application to send textual information to a Unicode application.

Additional reference words: 3.10

INF: Monitoring a Log File for an Event

Article ID: Q105301

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

To receive notification when something is written to the log file, use the following undocumented application programming interface (API):

```
UINT ElfChangeNotify( LogHandle, Event )
HANDLE LogHandle,
HANDLE Event
```

NOTE: This article was written to temporarily help customers who need this functionality. ElfChangeNotify() will be replaced by an officially documented API in the next revision of Windows NT.

This function will cause the Event to be signaled when an event has been written to the log file identified by the LogHandle.

LogHandle
Handle to a log file obtained from a call to OpenEventLog().

Event
A handle to a Win32 event.

Note that you must link with ADVAPI32.LIB.

Most of the ELF (Event Logging Facility) functions already have Win32 wrappers. ElfChangeNotify() does not have such a wrapper at this time, but will have a wrapper in the next release. At that time, it is recommended that code that uses ElfChangeNotify() be rewritten to use the new Win32 API, because ElfChangeNotify() may not be supported as is in the future.

Additional reference words: 3.10 logfile

BUG: Redirecting Output to an MS-DOS Application
Article ID: Q105303

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

The following method is used to redirect output to a child process when it is started from a GUI application:

1. Declare STARTUPINFO si.
 - a. Set the hStdIn, hStdOut, and/or hStdErr field as desired.
 - b. Set the dwFlags field to STARTF_USESTDHANDLES.
2. In CreateProcess(), set inherit handles to TRUE.

NOTE: This method does not work when starting MS-DOS applications.

RESOLUTION

=====

As a workaround, use AllocConsole(), then the following method:

1. Use SetStdHandle() to set the desired handles to be inherited.

-or-

Use DuplicateHandle() to change the inheritance property of handles that should not be inherited.

3. In CreateProcess(), set inherit handles to TRUE.

This method creates a blank console window; however, this is necessary because the method doesn't work otherwise.

STATUS

=====

Microsoft has confirmed this problem to be a bug in Windows NT 3.1.

MORE INFORMATION

=====

Note that if you are opening a handle that will be inherited by the child, set SECURITY_ATTRIBUTES.bInheritHandle = TRUE in the call to CreateFile(), CreatePipe(), and so forth.

For an example of both methods of redirection, see the INHERIT SDK

sample.

Additional reference words: 3.10

INF: SetErrorMode() Is Inherited

Article ID: Q105304

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

An application can use `SetErrorMode()` to control whether Windows handles serious errors or whether the application itself will handle the errors.

NOTE: The error mode will be inherited by any child process. However, the child process may not be prepared to handle the error return codes. As a result, the application may die during a critical error without the usual error message popups occurring.

This behavior is by design.

One solution is to call `SetErrorMode()` before and after the call to `CreateProcess()` in order to control the error mode that is passed to the child. Be aware that this process must be synchronized in a multithreaded application.

Additional reference words: 3.10

INF: Calling CRT Output Routines from a GUI Application

Article ID: Q105305

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

To use C run-time output routines, such as `printf()`, from a GUI application, it is necessary to create a console. The Win32 application programming interface (API) `AllocConsole()` creates the console. The CRT routine `setvbuf()` removes buffering so that output is visible immediately.

This method works if the GUI application is run from the command line or from File Manager. However, this method does not work if the application is started from the Program Manager or via the "start" command. The following code shows how to work around this problem:

```
int hCrt;
HFILE *hf;

AllocConsole();
hCrt = _open_osfhandle(
    (long) GetStdHandle(STD_OUTPUT_HANDLE),
    _O_TEXT
);
hf = _fdopen( hCrt, "w" );
*stdout = *hf;
i = setvbuf( stdout, NULL, _IONBF, 0 );
```

This code opens up a new low-level CRT handle to the correct console output handle, associates a new stream with that low-level handle, and replaces `stdout` with that new stream. This process takes care of functions that use `stdout`, such as `printf()`, `puts()`, and so forth. Use the same procedure for `stdin` and `stderr`.

Note that this code does not correct problems with handles 0, 1, and 2. In fact, due to other complications, it is not possible to correct this, and therefore it is necessary to use stream I/O instead of low-level I/O.

MORE INFORMATION

=====

When a GUI application is started with the "start" command, the three standard OS handles `STD_INPUT_HANDLE`, `STD_OUTPUT_HANDLE`, and `STD_ERROR_HANDLE` are all "zeroed out" by the console initialization routines. These three handles are replaced by valid values when the GUI application calls `AllocConsole()`. Therefore, once this is done, calling `GetStdHandle()` will always return valid handle values. The

problem is that the CRT has already completed initialization before your application gets a chance to call AllocConsole(); the three low I/O handles 0, 1, and 2 have already been set up to use the original zeroed out OS handles, so all CRT I/O is sent to invalid OS handles and CRT output does not appear in the console. Use the workaround described above to eliminate this problem.

In the case of starting the GUI application from the command line without the "start" command, the standard OS handles are NOT correctly zeroed out, but are incorrectly inherited from CMD.EXE. When the application's CRT initializes, the three low I/O handles 0, 1, and 2 are initialized to use the three handle numbers that the application inherits from CMD.EXE. When the application calls AllocConsole(), the console initialization routines attempt to replace what the console initialization believes to be invalid standard OS handle values with valid handle values from the new console. By coincidence, because the console initialization routines tend to give out the same three values for the standard OS handles, the console initialization will replace the standard OS handle values with the same values that were there before--the ones inherited from CMD.EXE. Therefore, CRT I/O works in this case.

It is important to realize that the ability to use CRT routines from a GUI application run from the command line was not by design so this may not work in future versions of Windows NT. In a future version, you may need the workaround not just for applications started on the command line with "start <application name>", but also for applications started on the command line with "application name".

Additional reference words: 3.10

INF: Getting and Using a Handle to a Directory

Article ID: Q105306

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

CreateDirectory() can be used to open a new directory. An existing directory can be opened by calling CreateFile(). To open an existing directory with CreateFile(), it is necessary to specify the flag FILE_FLAG_BACKUP_SEMANTICS. The following code shows how this can be done:

```
HANDLE hFile;

hFile = CreateFile( "c:\\mstools",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_WRITE,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_BACKUP_SEMANTICS,
    NULL
);
if( hFile == INVALID_HANDLE_VALUE )
    MessageBox( NULL, "CreateFile() failed", NULL, MB_OK );
```

The handle obtained can be used to obtain information about the directory or to set information about the directory. For example:

```
BY_HANDLE_FILE_INFORMATION fiBuf;
FILETIME ftBuf;
SYSTEMTIME stBuf;
char msg[40];

GetFileInformationByHandle( hFile, &fiBuf );
FileTimeToLocalFileTime( &fiBuf.ftLastWriteTime, &ftBuf );
FileTimeToSystemTime( &ftBuf, &stBuf );
wsprintf( msg, "Last write time is %d:%d %d/%d/%d",
    stBuf.wHour, stBuf.wMinute, stBuf.wMonth, stBuf.wDay, stBuf.wYear );
MessageBox( NULL, msg, NULL, MB_OK );
```

Additional reference words: 3.10

INF: The Use of PAGE_WRITECOPY
Article ID: Q105532

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The documentation indicates that the PAGE_WRITECOPY protection gives copy-on-write access to the committed region of pages. As it is, PAGE_WRITECOPY makes sense only in the context of file mapping, where you want to map something from the disk into your view and then modify the view without causing the data to go on the disk.

The only case where VirtualAlloc() should succeed with PAGE_WRITECOPY is the case where CreateFileMapping() is called with -1 and allocates memory with the SEC_RESERVE flag and later on, VirtualAlloc() is used to change this into MEM_COMMIT with a PAGE_WRITECOPY protection.

There is a bug in Windows NT 3.1 such that the following call to VirtualAlloc() will succeed:

```
lpCommit = VirtualAlloc(lpvAddr, cbSize, MEM_COMMIT, PAGE_WRITECOPY);
```

NOTE: lpvAddr is a pointer to memory that was allocated with MEM_RESERVE and PAGE_NOACCESS.

One case where this might be useful is when emulating the UNIX fork command. Emulating fork behavior would involve creating instance data and using threads or multiple processes.

Additional reference words: 3.10

BUG: Problems with Local/Global Memory Management APIs
Article ID: Q105533

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

The following is a list of problems that may be encountered when the Local and Global memory management application programming interfaces (APIs) are used:

1. The documentation for LocalUnlock() says that if the specified memory is fixed, the API returns nonzero.

The API returns 0 (zero) even if the memory is fixed.

2. The documentation for LocalReAlloc() states that if the cbBytes parameter is 0, the memory is movable and discardable, and the lock count is zero, then the API returns a handle to a discarded memory object. If the lock count is nonzero, the API will fail.

The API succeeds even if the lock count is nonzero.

3. The documentation for LocalFree() and for GlobalFree() indicates that if the parameter is NULL, the function will fail and the system generate an access violation.

The APIs do not generate an access violation.

4. The documentation for LocalFree() and for GlobalFree() states that the API will fail if passed a handle to a memory object that is locked.

The APIs free the memory even if the lock count is nonzero.

RESOLUTION/STATUS

=====

The following are confirmations of the corresponding problems described in the previous section:

1. This is an error in the documentation. Memory that is already unlocked will cause LocalUnlock() to return FALSE and GetLastError() will report ERROR_NOT_LOCKED. Memory allocated with LMEM_FIXED always has a lock count of zero, and therefore GetLastError() will also return ERROR_NOT_LOCKED in this case.

2. This is a bug in Local/GlobalReAlloc().

3. This is an error in the documentation. The Help should state that Local/GlobalFree() will return NULL if an attempt is made to free NULL. This is compatible with Windows 3.1.

4. This is a bug in Local/GlobalFree().

Additional reference words: 3.10

BUG: AllocConsole() Does Not Set Error Code on Failure
Article ID: Q105564

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

The documentation for AllocConsole() states the following:

If the function succeeds, the return value is TRUE; otherwise, it is FALSE. To get extended error information, use the GetLastError() function.

Upon failure, AllocConsole() does not set the error code.

STATUS

=====

Microsoft has confirmed this to be a bug in Windows NT 3.1.

MORE INFORMATION

=====

The following sample code demonstrates the problem:

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

void main()
{
    BOOL bSuccess;

    /* Comment out the following line and GetLastError() will    */
    /* return 999; otherwise, GetLastError() returns 1812.        */

    FreeConsole();

    SetLastError( 999 );

    bSuccess = AllocConsole();
    if( !bSuccess )
        puts( "AllocConsole failed" );
    printf( "The last error is: %d\n", GetLastError() );
    getchar();
}
```

Additional reference words: 3.10

INF: Critical Sections Versus Mutexes

Article ID: Q105678

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Critical sections and mutexes provide synchronization that is very similar, except that critical sections can be used only by the threads of a single process. There are two areas to consider when choosing which method to use within a single process:

1. Speed. The Synchronization overview says the following about critical sections:

... critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization.

Critical sections use a processor-specific test and set instruction to determine mutual exclusion.

2. Deadlock. The Synchronization overview says the following about mutexes:

If a thread terminates without releasing its ownership of a mutex object, the mutex is considered to be abandoned. A waiting thread can acquire ownership of an abandoned mutex, but the wait function's return value indicates that the mutex is abandoned.

WaitForSingleObject() will return WAIT_ABANDONED for a mutex that has been abandoned. However, the resource that the mutex is protecting is left in an unknown state.

There is no way to tell whether a critical section has been abandoned.

Additional reference words: 3.10

PRB: GetPrivateProfileSection() Can Read Only 32K Sections
Article ID: Q105681

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

The documentation for GetPrivateProfileSection() indicates that the application programming interface (API) can read all the keys and values of a section, regardless of size. However, GetPrivateProfileSection() seems to handle only sections that are smaller than 32K, even though the size of the buffer is a DWORD.

CAUSE

=====

The code is casting this value to a signed short, and therefore the problems with sections that are greater than 32K in size.

STATUS

=====

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

Additional reference words: 3.10

INF: Using NTFS Alternate Data Streams

Article ID: Q105763

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The documentation for the NTFS file system states that NTFS supports multiple streams of data; however, the documentation does not address the syntax for the streams themselves.

The Windows NT Resource Kit documents the stream syntax as follows:

```
filename:stream
```

Alternate data streams are strictly a feature of the NTFS file system and may not be supported in future file systems. However, NTFS will be supported in future versions of Windows NT.

Future file systems will support a model based on Object Linking and Embedding (OLE) version 2.0 structured storage (IStream and IStorage). By using OLE 2.0, an application can support multiple streams on any file system and all supported operating systems (Windows, Macintosh, Windows NT, and Win32s), not just Windows NT.

MORE INFORMATION

=====

The following sample code demonstrates NTFS streams:

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

void main( )
{
    HANDLE hFile, hStream;
    DWORD dwRet;

    hFile = CreateFile( "testfile",
                      GENERIC_WRITE,
                      FILE_SHARE_WRITE,
                      NULL,
                      OPEN_ALWAYS,
                      0,
                      NULL );
    if( hFile == INVALID_HANDLE_VALUE )
        printf( "Cannot open testfile\n" );
    else
        WriteFile( hFile, "This is testfile", 16, &dwRet, NULL );
}
```

```

hStream = CreateFile( "testfile:stream",
                     GENERIC_WRITE,
                     FILE_SHARE_WRITE,
                     NULL,
                     OPEN_ALWAYS,
                     0,
                     NULL );
if( hStream == INVALID_HANDLE_VALUE )
    printf( "Cannot open testfile:stream\n" );
else
    WriteFile( hStream, "This is testfile:stream", 23, &dwRet, NULL );
}

```

The file size obtained in a directory listing is 16, because you are looking only at "testfile", and therefore

```
type testfile
```

produces the following:

```
This is testfile
```

However

```
type testfile:stream
```

produces the following:

```
The filename syntax is incorrect
```

In order to view what is in testfile:stream, use:

```
more < testfile:stream
```

```
-or-
```

```
mep testfile:stream
```

Additional reference words: 3.10

INF: RegSaveKey() Requires SeBackupPrivilege
Article ID: Q106383

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The description for RegSaveKey() states the following:

The caller of this function must possess the SeBackupPrivilege security privilege.

This means that the application must explicitly open a security token and enable the SeBackupPrivilege. By granting a particular user the right to back up files, you give that user the right only to gain access to the security token (that is, the token is not automatically created for the user but the right to create such a token is given). You must add additional code to open the token and enable the privilege.

The following code demonstrates how this can be done:

```
static HANDLE          hToken;
static TOKEN_PRIVILEGES tp;
static LUID           luid;

// Enable backup privilege.

OpenProcessToken( GetCurrentProcess(),
    TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken );
LookupPrivilegeValue( NULL, "SeBackupPrivilege", &luid );
tp.PrivilegeCount      = 1;
tp.Privileges[0].Luid  = luid;
tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
AdjustTokenPrivileges( hToken, FALSE, &tp,
    sizeof(TOKEN_PRIVILEGES), NULL, NULL );

// Insert your code here to save the registry keys/subkeys.

// Disable backup privilege.

AdjustTokenPrivileges( hToken, TRUE, &tp,
    sizeof(TOKEN_PRIVILEGES), NULL, NULL );
```

MORE INFORMATION

=====

Note that you cannot create a process token; you must open the existing process token and adjust its privileges.

The DDEML Clock sample has similar code sample at the end of the CLOCK.C file where it obtains the SeSystemTimePrivilege so that it can set the system time.

Additional reference words: 3.10

INF: Identifying a Previous Instance of an Application

Article ID: Q106385

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The entry point of both Windows and Windows NT applications is documented to be:

```
int WinMain( hInstance, hPrevInstance, lpszCmdLine, nCmdShow )

HINSTANCE hInstance;      /* Handle of current instance */
HINSTANCE hPrevInstance;  /* Handle of previous instance */
LPSTR lpszCmdLine;        /* Address of command line */
int nCmdShow;             /* Show state of window */
```

However, under Windows NT, `hPrevInstance` is documented to always be `NULL`. The reason is that each application runs in its own address space and may have the same ID as another application.

To determine whether another instance of the application is running, use a named mutex. If opening the mutex fails, then there are no other instances of the application running. `FindWindow()` can be used with the class and window name. However, note that a second instance of the application could be started, and could execute the `FindWindow()` call before the first instance has created its window. Use a named object to ensure that this does not happen.

MORE INFORMATION

=====

The fact that `hPrevInstance` is set to `NULL` simplifies porting Win16 applications. Most Win16 applications contain the following logic:

```
if( !hPrevInstance )
    if( !InitApplication(hInstance) )
        return FALSE;
```

Under Windows, window classes only are registered by the first instance of an application. Consequently, if `hPrevInstance` is not `NULL`, then the window classes have already been registered and `InitApplication()` is not called.

Under Windows NT, because `hPrevInstance` is always `NULL`, `InitApplication()` is always called, and each instance of an application will correctly register its window classes.

Additional reference words: 3.10

Sample: Saving/Loading Bitmaps in .DIB Format on MIPS
Article ID: Q85844

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

In Win32, saving or loading a bitmap in .DIB file format is basically the same as in Win16. However, care must be taken in DWORD alignment, especially on the MIPS platform.

An exception occurs when loading or saving a bitmap on the MIPS platform. In NTSD, the following error message is received:

data mis-alignment

CAUSE

A non-DWORD aligned actual parameter was passed to a function such as GetDIBits().

The .DIB file format contains the BITMAPFILEHEADER followed immediately by the BITMAPINFOHEADER. Notice that the BITMAPFILEHEADER is not DWORD aligned. Thus, the structure that follows it, the BITMAPINFOHEADER, is not on a DWORD boundary. If a pointer to this DWORD misaligned structure is passed to the sixth argument of GetDIBits(), an exception will occur.

RESOLUTION

To resolve this problem, copy the data in the structure over to a DWORD-aligned memory and pass the pointer to the latter structure to the function instead. See the sample code LOADBMP.C for detail.

More Information:

There is a sample to illustrate this process. Refer to the LOADBMP.C file in the MANDEL sample.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

Sample: World Coordinate Transform

Article ID: Q81721

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SDK sample named WXFORM provides a demonstration of the new world-coordinate transformation. This sample displays a rectangle in world coordinates and a matrix containing the transform values. Users can directly manipulate the rectangle and see the effect on the transformation, or they can set the transformation and see the effect on the rectangle.

More Information:

The program begins by setting the viewport origin to the center of the client area. It then draws a rectangle in world coordinate space from the point (0, 0) to the point (100, 100). The user can directly manipulate this rectangle by using the left and right mouse buttons. Specific actions are described more fully in the "Direct Manipulation Help" dialog box.

There is a second dialog box titled "World Transform." This shows the values of the eM11, eM12, eM21, eM22, eDx, and eDy fields in the XFORM structure retrieved by calling the GetWorldTransform function. By choosing the buttons on this dialog box, the user can cause a SetWorldTransform to occur in the program.

There are three coordinate systems of interest in this sample. The first one is the world coordinate system, which is new to Win32. These points are ultimately mapped to the second coordinate system, device coordinates, before being painted in the window. This program must also use a third coordinate system, screen coordinates, for certain interactions with the mouse pointer.

There is a third dialog box titled "Mouse Position" that shows the location of the cursor in all three of these coordinate systems. The device coordinates are relative to the upper-left corner of the client area. They are not relative to the viewport origin.

Additional reference words: `ModifyWorldTransform`

Sample: AngleArc Demonstration Program

Article ID: Q81724

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The sample named ANGLE provides a demonstration of how the new AngleArc API function works. The X, Y, and RADIUS parameters are all in the world coordinate space. The start angle and sweep angle are floating-point values and are interpreted as degrees.

More Information:

This program presents a dialog box stretched across the top of the window. The user can set the parameters for the AngleArc API function by changing the values in the entry fields of this dialog box. A button on the dialog box then allows the user to immediately see the results of these values on the arc in the client area. If the values in the entry field are invalid, the program will write out this information and not draw the arc. The origin of the viewport is shifted down in the client area so that it exists at the upper-left corner of the viewable area.

Sample: Using GetDIBits() for Retrieving Bitmap Information
Article ID: Q85846

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When saving a bitmap in .DIB file format, the GDI function is used to retrieve the bitmap information. The general use of this function and the techniques for saving a bitmap in .DIB format are largely unchanged; however, this article provides more details on the use of the Win32 API version of the GetDIBits() function. MANDEL is a sample program that illustrates the information in this article.

More Information:

The function can be used to retrieve the following information:

- Data in the BitmapInfoHeader (no color table and no bits)
- Data in the BitmapInfoHeader and the color table (no bits)
- All the data (BitmapInfoHeader, color table, and the bits)

The fifth and the sixth parameters of the function are used to tell the graphics engine exactly what the application wants it to return. If the fifth parameter is NULL, then no bits will be returned. If the biBitCount is 0 (zero) in the sixth parameter, then no color table will be returned. In addition, the biSize field of the BitmapInfoHeader must be set to either the size of BitmapInfoHeader or BitmapCoreHeader for the function to work properly.

Refer to the SAVEBMP.C file in the MANDEL sample for details.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

Sample: Demonstration of Using System Info API
Article ID: Q81849

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The GETSYS SDK sample is a dialog box that provides the user with an easy way to see the results of the following API functions:

```
GetSysColors()  
GetSystemDirectory()  
GetSystemInfo()  
GetSystemMetrics()  
GetSystemPaletteEntries()  
GetSystemTime()
```

Sample: StretchBlt Demonstration

Article ID: Q81850

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The STREBLT sample is an easy to use demonstration of the StretchBlt API function. The program presents a dialog box on the top of the window, and through this dialog box the user can manipulate the parameters to StretchBlt. In the main window, the source bitmap is displayed on the right half of the window, and the destination bitmap is displayed on the left half.

More Information:

The source and destination rectangles may be changed directly in the dialog, or they may be changed by using the direct manipulation objects in the two halves of the window. Clicking and dragging the mouse in the upper-left corner moves the rectangles; clicking and dragging the mouse in the lower-right corner sizes the rectangles. The source direct manipulation object is temporarily erased before calling StretchBlt so that the top and left edges do not show in the destination image.

The raster operation for the StretchBlt call may be changed by altering the values in the right-most entry fields. The contents are interpreted to be in hexadecimal. There is a combo box directly beneath these entry fields that lists all of the standard raster operations. If the user selects a standard ROP from this combo box, its contents are copied into the ROP entry fields and are then used in the StretchBlt call.

Several of the raster operations make use of a pattern in the destination HDC. For this reason, the program also allows the user to select one of the standard pattern brushes from a second combo box. This brush is selected into the destination HDC just prior to making the StretchBlt call.

The effect of the StretchBlt call is also affected by the "StretchBlt mode" that has been set for the destination HDC. A third combo box allows the user to select from any of the standard modes. The difference is most easily observed when stretching from a large source rectangle to a small destination rectangle.

The "Draw" button may be chosen at any time to cause the StretchBlt call to be made. This does not erase the background, so that the effect of multiple ROPs on the HDC can be observed. Manipulating the source rectangle also causes a StretchBlt to occur without erasing the window. However, manipulating the destination rectangle erases the destination half of the window before the next StretchBlt is called.

Sample: Using Region-Related API Functions

Article ID: Q81874

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The REGIONS sample demonstrates various region-related API functions, and allows a user to create rectangular, elliptic, and multi-polygon regions. In addition, hit-testing a region and combining regions using the different region-combination modes is demonstrated.

More Information:

When the program has started, create a region by choosing one of the items in the Create submenu. At this point, items in the Options submenu will be enabled, and hit-testing, inversion, and other actions can be performed on the region.

When a second region is created, items in the Combine submenu will be enabled. Choosing one of these items causes CombineRgn to be called with the specified combine mode, and the two regions are merged into one.

It is possible to create up to three regions at a time. Items in the Options submenu always apply to the most recently created (or combined) region. The Erase item deletes all existing regions. Items in the Combine submenu always apply to the two most recently created (or combined) regions.

Additional reference words:

PtInRegion, CreateEllipticRgn, GetRgnBox, CreatePolygonRgn, CreateRectRgn, SetRectRgn, OffsetRgn, FillRgn, FrameRgn

Sample: PlgBlt Demonstration

Article ID: Q81875

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Win32 offers a new API function that will copy a bit image onto an arbitrary parallelogram. Now, for the first time, application programs can trivially rotate or shear bitmaps. The PLGBLT SDK sample is an easy to use demonstration of how this new API function may be used.

More Information:

The program presents a dialog box on top of the window that displays the input parameters to the PlgBlt function. By choosing the "New Src" or "New Mask" button, the user can select a new bitmap for use as the source bitmap or as the monochrome mask bitmap. The client area of the window is divided into three regions. The region on the left contains the result of the PlgBlt operation. The region in the middle provides the source HDC, and the region on the right provides the mask bitmap for the PlgBlt operation.

In each of the three regions, there is a "direct manipulation object." This object may be picked up and moved by clicking the left mouse button in the top-left corner and dragging. The three objects are restricted in their response to user actions to correctly reflect the parameters to the PlgBlt function. The object in the mask region may be moved only. The object in the source region may be moved or sized. The object in the destination region may be moved, sized, sheared, or rotated. Please see the WXFORD sample for more information on how this direct manipulation is accomplished. Additional information on the WXFORD sample may be obtained by querying on the word WXFORD in this knowledge base.

SAMPLE: Using Graphic Paths Demonstration

Article ID: Q81876

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The PATHS sample demonstrates the use of paths for drawing, filling, and clipping. The program draws six different figures in the window and labels each one. Each figure is based on the same path re-created six times. The six figures are the result of calling the following Windows functions (with the poly fill mode in parentheses):

```
StrokePath()  
FillPath()  
StrokeAndFill() (Winding)  
StrokeAndFill() (Alternate)  
SelectClipPath() (Winding)  
SelectClipPath() (Alternate)
```

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

Additional reference words: 3.10

SAMPLE: PolyBezier() Demonstration

Article ID: Q81877

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The BEZIER sample provides an easy to use demonstration of how the PolyBezier() function works. The user can place points in the window with the left mouse button. The user can also move these points by dragging with the same mouse button. The PolyBezier() curve is drawn dynamically to follow the position of the new points.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

In order to use this program, press the left mouse button at miscellaneous places in the client area. A Polyline() call shows exactly where the points were put. When there are 4, 7, 10, ..., (3n+1) points on the screen, the PolyBezier() curve is drawn with these as control points. The API itself does not draw anything if there are some other number of points. The whole client area may be erased by pressing the right mouse button.

Additional reference words: 3.10

Sample: GetDeviceCaps() Demonstration Program
Article ID: Q83930

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The GETDEV sample is a dialog box that displays the result of the GetDeviceCaps call with all of the possible input parameters. Six of the numeric results (TECHNOLOGY, LINECAPS, POLYGONALCAPS, TEXTCAPS, CLIPCAPS, and RASTERCAPS) are expanded to show the constant string from WINGDI.H.

Sample: PolyDraw Function Demonstration

Article ID: Q83931

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The POLYDRAW sample provides an easy-to-use demonstration of how the PolyDraw Win32 API function works. The user can place points in the window with the left mouse button, and move these points by dragging with the same mouse button. The PolyDraw curve is drawn dynamically to follow the position of the new points.

More Information:

To use this program, click the left mouse button at miscellaneous places in the client area. A Polyline call shows exactly where the points were put. By default, the type entered into the type array is PT_LINETO. This can be changed to a PT_MOVETO type by holding down the SHIFT key. It can be changed to a PT_BEZIERTO type by holding down the CTRL key. The resulting purple curve shows the results. There will be no curve when the bezier points do not come in groups of 4, 7, ... , (3n+1).

INF: Use 16-Bit .FON Files for Cross-Platform Compatibility
Article ID: Q100487

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The AddFontResource() function installs a font resource in the GDI font table. Under Windows NT, the module can be a .FON file or a .FNT file. Under Windows 3.1, the module must be a .FON file. When using Win32s, AddFontResource() passes its argument to the Win16 AddFontResource, and therefore .FON files should be used for portability.

In addition, when running under Windows NT, the module can be either a 32-bit "portable executable" or a 16-bit .FON file. However, if the same Win32 executable is run under Win32s, the call to AddFontResource() fails if the *.FON is not in 16-bit format. Therefore, for compatibility across platforms, use 16-bit *.FON files. These can be created using the Windows 3.1 Software Development Kit (SDK).

Additional reference words: 3.10

Sample: MaskBlt Function Demonstration

Article ID: Q84541

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MASKBLT sample is an easy-to-use demonstration of the MaskBlt Win32 API function. The program presents a dialog box on the top of the window; through this dialog box the user can manipulate the parameters to MaskBlt. In the main window, the source bitmap is displayed in the center third of the window, the monochrome bitmap mask is displayed in the right third of the window, and the destination bitmap is displayed on the left.

More Information:

The destination rectangle may be changed directly in the dialog box, or it may be changed by using the direct manipulation object in the left third of the window. Clicking and dragging the mouse in the upper-left corner moves the rectangle; clicking and dragging the mouse in the lower-right corner sizes the rectangle. The function requires only a starting point (not a rectangle) for the source and mask bitmaps. There is one additional direct manipulation object for the source and one for the mask. These objects may be moved by clicking and dragging with the mouse.

The raster operation for the MaskBlt call may be changed by altering the values in the right most entry fields. The contents are interpreted to be in hexadecimal. There is a combo box directly beneath these entry fields that lists all of the standard raster operations. If the user selects a standard ROP from this combo box, its contents are copied into the ROP entry fields and are then used in the MaskBlt call.

This sample provides clipboard support in the following manner. Hitting <ctrl><insert> will copy the destination image into the clipboard. Hitting <shift><insert> will copy a bitmap from the clipboard into the source region. Hitting <alt><insert> will do both; the destination image will be copied into the clipboard and then down to the source region.

INF: Device Contexts: Using Across Threads

Article ID: Q94236

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A window created with the CS_OWNDC style retains its device context (DC) attributes across GetDC() calls.

However, the DC attributes are not retained if the GetDC() calls are called from different threads. This is by design because DCs are thread-based. In the Win32 user interface, if the calling thread is not the owner of the window, then GetDC() returns a cache DC instead of the owned DC handle.

To save attributes across threads, one must create a routine to initialize DC attributes, which is then called from threads not owning the given window.

Additional reference words: 3.10 3.1

INF: Transparent Blts in Windows NT
Article ID: Q89375

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In order to perform a transparent blt in Microsoft Windows versions 3.0 and 3.1, the BitBlt() function must be called two or more times. This process involves nine steps. (For more information on this process, see article Q79212 in the Microsoft Knowledge Base.)

Windows NT introduces a new method of achieving transparent blts. This method involves the use of the MaskBlt() function. The MaskBlt() function lets you use any two arbitrary ROP3 codes (say, SRCCOPY and BLACKNESS) and apply them on a pel-by-pel basis using a mask.

More Information:

For this example, the source and target bitmaps contain 4 BPP. The call to the MaskBlt() function is as follows:

```
MaskBlt(hdcTrg, // handle of target DC
        0, // x coord, upper-left corner of target rectangle
        0, // y coord, upper-left corner of target rectangle
        15, // width of source and target rectangles
        15, // height of source and target rectangles
        hdcSrc, // handle of source DC
        0, // x coord, upper-left corner of source rectangle
        0, // y coord, upper-left corner of source rectangle
        hbmMask, // handle of monochrome bit-mask
        0, // x coord, upper-left corner of mask rectangle
        0, // y coord, upper-left corner of mask rectangle
        0xAACC0020 // raster-operation (ROP) code
    );
```

The legend is as follows

- '.' = 0,
- '@' = 1,
- '+' = 2,
- '*' = 3,
- '#' = 15

Source Bitmap	Mask Bitmap	Target Bitmap	Result
*****@.....	#####	#####*
*****@@@.....	#####	#####**
*****@@@@.....	##.....##	##...+***+...##


```

+++*****+++*****
...@.@.@.@.@... ##.....## ##..*+*+*+*+*..##
+++*****+++***** ..@.@.@.@.@.@.@... ##.....## ##.*****+*+*+*+*..##
+++*****+++***** .@.@.@.@.@.@.@.@.@... ##.....## ##+*****+*+*+*+*+*##
*****+*+*+*+*+*+*+*+* .@.@.@.@.@.@.@.@.@.@... ##.....## ##*+*+*+*+*+*+*+*+*##
*****+*+*+*+*+*+*+*+* @.@.@.@.@.@.@.@.@.@.@... ##.....## ##*+*+*+*+*+*+*+*+*##
*****+*+*+*+*+*+*+*+* .@.@.@.@.@.@.@.@.@.@... ##.....## ##*+*+*+*+*+*+*+*+*##
+++*****+++***** ..@.@.@.@.@.@.@.@.@... ##.....## ##+*****+*+*+*+*+*+*##
+++*****+++***** ...@.@.@.@.@.@.@.@... ##.....## ##.*****+*+*+*+*..##
+++*****+++***** .....@.@.@.@... ##.....## ##..*+*+*+*+*..##
*****+*+*+*+*+*+*+*+* .....@.@.@... ##.....## ##...+*+*+*+*...##
*****+*+*+*+*+*+*+*+* .....@.@... ##.....## ##*****+*+*+*+*+*+*+*+*##
*****+*+*+*+*+*+*+*+* .....@..... ##.....## ##*****+*+*+*+*+*+*+*+*##

```

Note that the ROP "AA" is applied where 0 bits are in the mask and the ROP "CC" is applied where 1 bit is in the mask. This a transparency.

When creating a ROP4, you can use the following macro:

```
#define ROP4(fore,back) (((back) << 8) & 0xFF000000) | (fore))
```

This macro can be used to call the MaskBlt() function as follows:

```
MaskBlt(hdcDest, xTrgt, yTrgt,
        cx, cy,
        hdcSrc, xSrc, ySrc,
        hbmMask, xMask, yMask,
        ROP4(PATCOPY, NOTSRCCOPY)
        );
```

This call would draw the selected brush where 1 bit appears in the mask and bitwise negation of the source bitmap where 0 bits appear in the mask.

Additional reference words: 3.10 pixel

INF: 16 and 32 Bits-Per-Pel Bitmap Formats

Article ID: Q94326

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Win32 supports the same bitmap formats as Microsoft Windows version 3.1, but includes two new formats: 16 and 32 bits-per-pel.

For DIBs (device independent bitmaps), the 16- and 32-bit formats contain three DWORD masks in the bmiColors member of the BITMAPINFO structure (the color table), instead of the array of RGBQUAD structures for 1-, 4-, and 8-bit formats). These masks specify which bits in the pel correspond to which color.

The three masks must have contiguous bits, and their order is assumed to be R, G, B (high bits to low bits). The order of the three masks in the color table must also be first red, then green, then blue (RGB). In this manner, the programmer can specify a mask indicating how many shades of each RGB color will be available for bitmaps created with CreateDIBitmap().

For 16-bit bitmaps, CreateBitmap() defaults to the 5+5+5 format. The formats 6+5+5 and 6+6+4 are also supported by this method.

Example

The 5-5-5 format masks are:

```
0x00007C00  red   (0000 0000 0000 0000 0111 1100 0000 0000)
0x000003E0  green (0000 0000 0000 0000 0000 0011 1110 0000)
0x0000001F  blue  (0000 0000 0000 0000 0000 0000 0001 1111)
```

For 16 bits-per-pel, the upper half of the DWORDs are always zeroed.

Usage

When using 16- and 32-bit formats, there are also certain fields of the BITMAPINFOHEADER structure that must be set to the correct values:

1. The biCompression member must be set to BI_BITFIELDS, or the DIB API will fail. This indicates that there are bit fields in the bitmap rather than encoded color indexes (as in 4- or 8-bit formats).
2. The biClrUsed member must be set to 3 (or 0, which means use the maximum number of color table entries for that format, which for 16 and 32 bits-per-pel is 3). The member names are a bit misleading in these cases, because the "color table" (the bmiColors entry of the BITMAPINFO

structure) is not being used to store a list of colors, but 3 bit masks.

A technical note related to this subject from the Microsoft Multimedia group is also available. It can be obtained from CompuServe in the WINEXT and WINSKD forums. The filename is VFW.ZIP. In addition, the technote is available by calling Microsoft Developer Services at (800) 227-4679, extension 11771. The technical note is part of the Video for Windows technical notes and describes how to create a display driver that supports these new DIB formats, which are used by Video for Windows. The technical note also includes definitions of installable image codecs.

Additional reference words: 3.10 technote

INF: PSTR's in OUTLINETEXTMETRIC Structure

Article ID: Q90085

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The OUTLINETEXTMETRIC structure ends with four fields of type PSTR. The four fields in question are not actually absolute pointers. They are offsets from the beginning of the OUTLINETEXTMETRIC structure to the strings in question, as the documentation indicates:

otmpFamilyName

Specifies the offset from the beginning of the structure to a string specifying the family name for the font.

otmpFaceName

Specifies the offset from the beginning of the structure to a string specifying the face name for the font. (This face name corresponds to the name specified in the LOGFONT structure.)

otmpStyleName

Specifies the offset from the beginning of the structure to a string specifying the style name for the font.

otmpFullName

Specifies the offset from the beginning of the structure to a string specifying the full name for the font. This name is unique for the font and often contains a version number or other identifying information.

The only difference between this structure in Windows 3.1 and Windows NT is that the strings may be stored in either Unicode or ASCII under NT.

Additional reference words: 3.10

INF: Advantages of Device-Dependent Bitmaps

Article ID: Q94918

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A DDB (device-dependent bitmap) is much faster than a DIB (device independent-bitmap) to BitBlt(). For this reason, it is often a good strategy under Win32 (as well as under Windows 3.1) to create a DDB from a DIB when caching or calling *Blt() functions.

The slight drawback of memory overhead for the DDB is handled well by Win32. Under Windows 3.1, the DDB memory could be marked as discardable. Under Win32, the memory will be paged out if system resources become tight (at least until the next repaint); if the memory is marked as PAGE_READONLY, it can be efficiently reused, [see VirtualProtect() in the Win32 application programming interface (API) Help file].

However, saving the DDB to disk as a mechanism for transfer to other applications or for later display (another invocation) is not recommended. This is because DDBs are driver and driver version dependent. DDBs do not have header information, which is needed for proper translation if passed to another driver or, potentially, to a later version of the driver for the same card.

Additional reference words: 3.10 3.1 win3.1

**INF: Set/ModifyWorldTransform() Requires SetGraphicsMode()
Article ID: Q94922**

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SetWorldTransform() and ModifyWorldTransform() fail if GM_ADVANCED mode is not previously set by calling SetGraphicsMode().

Likewise, if the graphics mode in the device context (DC) is set to GM_ADVANCED and the world transform is set to anything but the identity transform, the DC cannot be reset to the default mode (GM_COMPATIBLE) unless the world transform is first reset to the identity transform.

Thus, applications involving SetWorldTransform() or ModifyWorldTranform() should be changed so that SetGraphicsMode(hdc,GM_ADVANCED) is called first.

Additional reference words: 3.10 3.1

PRB: IsGdiObject() Is Not a Part of the Win32 API
Article ID: Q91072

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

There doesn't seem to be an IsGdiObject() function in Win32 in either the API reference or the header files.

CAUSE

The function was added to the Windows 3.1 API because passing a handle to a non-GDI object to a GDI function causes a GP fault under Windows 3.0. Win32 on Windows NT detects whether the APIs are passed an inappropriate handle, and the function returns an error.

RESOLUTION

IsGdiObject() is not needed for Win32 on Windows NT.

Additional reference words: 3.10 3.1 3.00 3.0

INF: Use of DocumentProperties() vs. ExtDeviceMode()

Article ID: Q92514

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Windows applications have used ExtDeviceMode() to retrieve or modify device initialization information for printer drivers. The Win32 API introduces a new function DocumentProperties() that applications can use to configure the settings of the printer.

Note that ExtDeviceMode() calls DocumentProperties(); therefore, it is faster for applications to use DocumentProperties() directly.

Specifying the DM_UPDATE mask allows an application to change printer settings when using DocumentProperties(). Applications should be aware that the GetProcAddress() function is now case sensitive.

Windows 3.x applications running on Windows NT (WOW) can call ExtDeviceMode(). The spooler's ExtDeviceMode() entry is intended for WOW use.

Additional reference words: 3.00 3.0 3.10 3.1

INF: Font-Related APIs & Structures Removed from Win32/NT
Article ID: Q93465

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following font-related APIs, structures, constants, and typedefs are being removed from Win32/NT. They are being removed due to schedule constraints and because they do not extend well to systems with large numbers of fonts installed. Additional design work will be done and a better solution provided in a future release.

APIs:

```
EnumNearestFont()  
SetFontMapperControls()  
GetFontMapperControls()
```

```
AddFontModule()  
RemoveFontModule()
```

Structures:

```
tagFMPENALTYSET  
tagFMWEIGHTSET  
tagFMATCH  
tagFMCONTROLS
```

Constants:

```
FMATCH_EXACT  
FMATCH_NEAR  
FMATCH_FAR  
FMATCH_ERROR  
PANOSE_RANGE  
FM_LOCATION_GDI  
MAPPER_INDEX_TERMINATE  
MAPPER_INDEX_HEIGHT  
MAPPER_INDEX_WIDTH  
MAPPER_INDEX_ESCAPEMENT  
MAPPER_INDEX_ORIENTATION  
MAPPER_INDEX_WEIGHT  
MAPPER_INDEX_ITALIC  
MAPPER_INDEX_UNDERLINE  
MAPPER_INDEX_STRIKEOUT  
MAPPER_INDEX_CHARSET  
MAPPER_INDEX_OUTPRECISION  
MAPPER_INDEX_CLIPPRECISION  
MAPPER_INDEX_QUALITY
```

MAPPER_INDEX_PITCHANDFAMILY
MAPPER_INDEX_FACENAME
MAPPER_INDEX_FULLNAME
MAPPER_INDEX_STYLE
MAPPER_INDEX_PANOSE
MAPPER_INDEX_VENDORID
MAPPER_INDEX_ASPECT
MAPPER_INDEX_LOCATION
MAPPER_INDEX_LAST
SIZEOFMAPORDER
SIZEOFFMCONTROLS

Typedefs:

FMORDER

Additional reference words: 3.10

INF: DEVMODE and dmSpecVersion

Article ID: Q96282

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The dmSpecVersion field of the DEVMODE structure is intended for printer driver use only; no application programs should test this field. The purpose of this field is for new printer drivers to be able to recognize and handle DEVMODE structures created according to previous DEVMODE structure specification.

The DEVMODE structure is used for printer and (occasionally) display drivers when initializing. This structure is tied to the driver--not the operating system. The dmSpecVersion field does not allow an application to determine which platform (Windows version 3.1, Windows on Windows, Win32) the application is running in.

When an application fills a DEVMODE structure, it should set the dmSpecVersion field to DM_SPECVERSION. This identifies the version of the DEVMODE structure the application is generating.

If the application is querying to understand an unknown device, then special attention should be paid to the dmFields, dmSize, and dmDriverExtra fields. These fields are a reliable means of understanding what fields in the DEVMODE structure are readable.

More Information:

The DEVMODE structure consists of public and private parts. The dmSpecVersion field applies to the public part. Any previously defined fields are not altered when the DEVMODE specification is updated--more fields are merely added to the end of the structure. This can mean fields used in the previous specification are ignored in a later specification. This functionality is managed by one bitfield describing what fields a driver actually uses. The new drivers just switch off the obsolete fields.

Applications using DEVMODE should always use the dmSize and dmDriverExtra fields for allocating/storing/manipulating the structure. These fields define the sizes of the public and private parts of the structure, respectively.

Additional reference words: 3.10 3.1 WOW

INF: Tracking Brush Origins in Windows NT
Article ID: Q102353

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

When programming for Windows NT, it is no longer necessary to keep track of brush origins yourself. GDI32 keeps track of the brush origins by automatically recognizing when the origin has been changed. In Windows, you have to explicitly tell GDI to recognize the change by calling `UnrealizeObject()` with a handle to the brush. When a handle to a brush is passed to `UnrealizeObject()` in Windows NT, the function does nothing.

In Windows 3.1, the default brush origin is the screen origin. In Windows NT, the default origin is the client origin.

Additional reference words: 3.10

INF: Calculating the TrueType Checksum

Article ID: Q102354

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

To calculate a TrueType checksum:

1. Sum all the ULONGS in the .TTF file, except the checksumAdjust field (which contains the calculated checksum). Note that TrueType files are big-endian, while Windows and Windows NT are little-endian, so the bytes must be swapped before they are summed.
2. Subtract the result from the magic number 0xb1b0afba.

Example

1. Open the SYMBOL.TTF distributed with Windows NT. It is 64492 bytes long.
2. Step through the 16123 ULONGS, summing each one, except for the checksumAdjust field for the file (which in this case is 0xa7a81151).
3. Subtract the result from 0xb1b0afba. The result is 0xa7a81151.

The TrueType font file specification is available from several sources, including the Microsoft Software/Data Library (query on the word TTSPEC1).

Additional reference words: 3.10

INF: Creating a Font for Use with the Console

Article ID: Q105299

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

It is possible to use FontEdit to create a font that can be used by the console. The following must be true:

- The face name must be System, Terminal, or Courier
- The font size must be different from any of the other console fonts
- The font must be fixed pitch
- The font must not be italic

In addition, in the U.S. market, the font should support codepage 437.

Install the font from the Control Panel. After rebooting, the font will be available to the console.

An EnumFonts() call is made by the console during its initialization to determine what fonts are available. The console saves a set of one-to-one mappings between the font sizes listed and a set of LOGFONTs. The console never has direct knowledge of what file is used.

Additional reference words: 3.10

INF: Creating a Logical Font with a Nonzero lfOrientation
Article ID: Q104010

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

To create a font that writes in a direction other than left to right, an application should specify a nonzero lfEscapement in the LOGFONT structure that is passed to CreateFontIndirect(). This method works under Windows NT regardless of the graphics mode of the device context.

To create a font where the characters themselves are rotated, the application should specify a nonzero lfOrientation in the LOGFONT structure that is passed to CreateFontIndirect(). However, this setting is ignored in Windows NT unless the graphics mode is set to GM_ADVANCED.

Therefore, to successfully create a logical font with a nonzero lfOrientation, use

```
SetGraphicsMode( hDC, GM_ADVANCED )
```

to set the graphics mode of the device context to GM_ADVANCED.

MORE INFORMATION

=====

The TTFONTS sample program is a good way to quickly and easily see the effects of the lfEscapement and lfOrientation fields. However, TTFONTS does not set the graphics mode of its test window HDC to GM_ADVANCED. As a result, the lfOrientation field apparently is ignored. It is easy to modify the DISPLAY.C module of TTFONTS in order to set the graphics mode of the window HDC to GM_ADVANCED.

Additional reference words: 3.10

INF: Windows Socket API Specification Version 1.1

Article ID: Q85965

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Windows Sockets API Specification version 1.1 is now available in two formats. WINSOCK.DOC is a Word for Windows, version 2.0, document and WINSOCK.TXT is an ASCII-text-format document.

These files contain the Windows Sockets API Specification version 1.1, which defines a standard binary interface for tcp/ip transports based on the Berkeley Sockets interface originally in Berkeley UNIX with Windows-specific extensions. This specification has been endorsed by 20 leading companies, and defines the sockets interface in Windows NT. The specification will be supported by various vendors in their upcoming tcp/ip product releases for Windows for MS-DOS.

The Windows Sockets API Specification version 1.1 can be found in the Software/Data Library by searching on the words WINSOCK (ASCII text) or WINSOCKW (Word for Windows), the Q number of this article, or S13474 for WINSOCK or S13475 for WINSOCKW. WINSOCK and WINSOCKW were archived using the PKware file-compression utility.

Additional reference words: 1.00 2.00 1.10

INF: Writing a Telnet Client

Article ID: Q95866

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Windows NT provides a Telnet client service, which allows developers to easily write applications that connect with hosts that have a Telnet daemon process running.

To use the Telnet client service, the workstation must have the TCP/IP protocol installed properly. To start the Telnet service, type

```
net start telnet
```

at a command shell prompt, or use the Services application in the Control Panel to start the Telnet service.

The interface to the Telnet service is through the Win32 communications APIs (application programming interfaces). Instead of opening "COM1:" or "COM2:", the special device name TELNET is opened. Handling the device is the same as handling a standard communications port (some functionalities are not supported, such as setting baud rates, DCB settings, and so forth).

For an example of how to use the Telnet client service, see the TTY sample in the Win32 Software Development Kit (SDK) under the \MSTOOLS\SAMPLES\TTY subdirectory.

More Information:

Before connecting to a remote host, the IP address of both the remote host and the local host must be entered properly in the hosts file, located in the \%SYSTEMROOT%\SYSTEM32\DRIVERS\ETC subdirectory.

Additional reference words: TTY TCPIP

INF: Supported Versions of Windows Sockets

Article ID: Q101377

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

When using the WSStartup() Windows Sockets application programming interface (API)

```
int PASCAL WSStartup ( WORD wVersionRequired, LPWSADATA lpWSADATA );
```

the first parameter specifies the version of Windows Sockets the application requires.

The 16-bit Windows Sockets dynamic-link library (DLL) for NT (WINSOCK.DLL) supports versions 1.0 and 1.1.

The 32-bit Windows Sockets DLL for NT (WSOCK32.DLL) supports only version 1.1. Specifying version 1.0 in the call to WSStartup() results in a return of WSAVERNOTSUPPORTED.

The Win32s WSOCK32.DLL thunks down to the 16-bit WINSOCK.DLL, if it is installed.

Additional reference words: 1.00 1.10 3.10

INF: Using RPC Callback Functions

Article ID: Q96781

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The standard remote procedure call (RPC) model has a server containing one or more exported function calls, and a client, which calls the server's exported functions. However, Microsoft's implementation of RPC defines callbacks as a special interface definition language (IDL) attribute allowing a server to call a client function.

Callbacks can be used only in the context of a server call. Thus, a server may call a client's callback function only when the server is performing a client's remote procedure call (before it returns from processing). For example:

CLIENT	SERVER
-----	-----
Client makes RPC call. -->	
	<--- Server calls callback procedure.
Client returns from callback. --->	
	<--- Server calls callback procedure.
Client returns from callback. --->	
	<--- Server returns from original RPC call.

More Information:

Callbacks are declared in the RPC .IDL file and defined in the source of the client. The following demonstrates how callbacks are declared and defined:

```
[ SAMPLE.IDL ]
[
    uuid(9FEE4F51-0396-101A-AE4F-08002B2D0065),
    version(1.0),
    pointer_default( unique )
]

{
    void RPCProc( [in, string] unsigned char *pszStr );
    [callback] void CallbackProc([in,string] unsigned char *pszStr);
}

[ SAMPLEC.C (Client)]
/*
Callback RPC call (initiated from server, executed on client).
*/
void CallbackProc( unsigned char *pszString )
```

```
{
    printf("Call from server, printed on client: %s", pszStr );
}
```

```
[ SAMPLES.C (Server)]
```

```
/*
```

```
"Standard" RPC call (initiated from client, executed on server).
```

```
Makes a call to client callback procedure, CallbackProc().
```

```
*/
```

```
void RPCProc( unsigned char *pszStr )
```

```
{
```

```
    printf("About to call Callback() client function.."
```

```
    CallbackProc( pszStr );
```

```
    printf("Called callback function.");
```

```
}
```

In the makefile for the sample, the "-ms_ext" switch must be used for the MIDL compile. For example:

```
midl -ms_ext -cpp_cmd $(cc) -cpp_opt "-E" sample.idl
```

Additional reference words: 3.10

PRB: RPC Installation Problem

Article ID: Q104315

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

There is a problem with the Remote Procedure Call (RPC) service installation. Setup searches the Windows directory for WINSOCK.DLL. If the file is found, Setup installs RPC16C3.DLL.

This is a problem if you are dual booting a machine between Windows 3.1 and Windows NT because Windows 3.1 uses an older TCP/IP interface, which is called RPC16C3X.DLL on the distribution disks. When you run Windows NT, you will want to use the newer TCP/IP interface, called RPC16C3.DLL.

WORKAROUND

=====

If you are dual booting and using RPC, a workaround to this problem is to rename your WINSOCK.DLL file to something else, such as WINSOCK.XXX. This will cause Setup to copy the correct version of the TCP/IP dynamic-link library (DLL).

Additional reference words: 3.10

PRB: AttachThreadInput() Resets Keyboard State

Article ID: Q100486

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

SYMPTOMS

Start with a program that calls AttachThreadInput() to a thread in another process. Call GetKeyboardState() to get the current key. Call SetKeyboardState() to set the keystate. This call returns TRUE, indicating success, but the keystate is not successfully set.

If the thread is in the same process, calling SetKeyboardState() works as expected.

CAUSE

When attaching to another thread, a temporary message queue is created. This queue contains a copy of the keystate information from the queue to which you are attaching. When the keystate is set, the temporary queue keystate is updated and the application programming interface (API) succeeds. However, when the detach occurs, the keystate change information is lost and reverts to what it was before the attach.

RESOLUTION

To work around the problem, either:

- Stay attached

-or-

- Use hooks

STATUS

This problem will not be resolved in the release of Windows NT version 3.1; however, a resolution is being considered for a future release.

Additional reference words: 3.10

INF: Global Classes in Win32

Article ID: Q80382

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Under 16-bit Windows (Win16), when an application wants to check whether or not a window class has been previously registered in the system, it typically checks `hPrevInstance`. Under Win32, `hPreviousInstance` always returns `FALSE`. Thus, the window class will be reregistered by all `.EXEs` using the DLL that contains the window class, unless the check for `hPrevInstance` is called from a process that is spawned by the process that first registered the class.

More Information:

A class is registered for a process context only. A class definition is not available outside the process context of the process that originally registered it.

Although an application can enumerate windows and retrieve the class name of a class registered in another process, they will not be able to use that class within their process.

Sample: Common Dialog DLL

Article ID: Q81703

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A sample demonstrating the use of all of the common dialog box functions in the Win32 API is now available.

More Information:

Each dialog box is demonstrated being used in three different ways: standard, using a hook function, and using a modified template.

Additional reference words:

ChooseColor, ChooseFont, GetOpenFileName, GetSaveFileName

SAMPLE: Standard DLL & Ex. of Creating a Custom Control Class
Article ID: Q81852

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The SPINCUBE sample provides a generic Windows NT dynamic link library (DLL) template demonstrating the use of DLL entry points, exported variables, using C run time in a DLL, and so forth.

This sample also provides a functional example of how to create a custom control class that may be used by applications (for exampl, SPINTEST.EXE) as well the Dialog Editor.

MORE INFORMATION

=====

SPINCUBE.DLL contains the control window procedure and the interface functions required by the Dialog Editor (see SPINCUBE.C), as well as the control paint routines (see PAINT.C). SPINTEST.EXE is a small test program that loads SPINCUBE.DLL and creates a few of the custom controls.

To test SPINCUBE with the Dialog Editor:

1. Start the editor. From the File menu, choose Open Custom.
2. Enter the path and filename of SPINCUBE.DLL.
3. Create a new dialog box and choose a custom control button from the control palette (lower-right corner).
4. Click the dialog box to create a SPINCUBE control.
5. Save the dialog box template.
6. Inspect the .DLG file that was created.

Additional reference words: 3.10

INF: DDEML Application-Instance IDs Are Thread Local
Article ID: Q94091

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When using the DDEML (Dynamic Data Exchange Management Library) libraries from a spawned thread, the application-instance ID that is returned in the `lpidInst` parameter of `DdeInitialize` is thread local.

Therefore, the application-instance ID cannot be used by any other thread that is spawned by the process, nor can it be inherited from the parent.

To use the DDEML libraries within a thread, it is necessary to make both the `DdeInitialize` call and to use the `DdeUninitialize` call from within the thread; otherwise, there is no way to terminate the DDEML session.

Additional reference words: 3.10 3.1

SAMPLE: WM_COMMNOTIFY Message is Obsolete

Article ID: Q94561

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Under Windows version 3.1, the WM_COMMNOTIFY message is posted by a communication device driver whenever a COM port event occurs. The message indicates the status of a window's input or output queue.

This message is currently not supported under Windows NT. By release, however, the WOW layer will support the EnableCommNotification() function.

To duplicate the Windows 3.1 functionality under Windows NT, refer to the TTY sample, included with the SDK. The TTY sample is a common code base sample, which uses the EnableCommNotification() API under Windows 3.1 to tell COMM.DRV to post messages to the TTY window.

Under NT, this behavior is simulated with a secondary thread by using WaitCommEvent() and PostMessage().

TTY.C defines WM_COMMNOTIFY if WIN32 is defined. Using this method, WM_COMMNOTIFY notifications are simulated but use the same message definition as Windows 3.1.

The TTY sample is located on the Win32 SDK CD in \MSTOOLS\SAMPLES\COMM.

Additional reference words: 3.10 win 3.1 win3.1

Sample: SUBCLASS Program Demonstration

Article ID: Q84242

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SUBCLASS sample demonstrates how a program can subclass standard controls in order to extend their normal functionality. This sample replaces the window procedure for buttons, edit fields, and list boxes.

More Information:

The standard subclassing technique is to replace the window procedure in the window structure by using:

```
SetWindowLong (hwnd, GWL_WNDPROC, (LONG) SubclassWndProc);
```

In the SUBCLASS sample, the old window procedure is also saved in a structure pointed at by the user data. Thus, any functionality can be added to various classes of windows without having to know what the class originally was.

In this sample, the subclass procedure adds the ability to move and size the control windows when the application is not in "test mode." When the application is in test mode, the subclass procedure calls the original window procedure and the controls behave as normal. Thus, this sample provides the bare bones for a "dialog editor" type of program.

Additional reference words: CallWindowProc GWL_USERDATA

Sample: Communications API Function Demonstration

Article ID: Q87331

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The COMM SDK sample application is designed to demonstrate the basics of using the Win32 communications API functions while maintaining a common code base with Win16 code.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

The COMM program performs communications using the Windows functions `OpenFile()`, `ReadFile()`, `SetCommState()`, `SetCommMask()`, `WaitCommEvent()`, `WriteFile()`, and `CloseFile()`.

This sample creates a background thread to watch for COMM receiver events and posts a notification message to the main terminal window. Foreground character processing is written to the communications port.

Simple TTY character translation is performed and a screen buffer is implemented for use as the I/O window.

Overlapped file I/O techniques are demonstrated.

How to Use

The baud rate, data bits, stop bits, parity, port, RTS/CTS handshaking, DTR/DSR handshaking, and XON/XOFF handshaking can be changed under the Settings menu item.

Once the communications settings are set up, the Action menu item can be selected to connect or disconnect the TTY program.

INF: Freeing PackDDElParam() Memory

Article ID: Q94149

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When posting DDE messages via `PostMessage()`, an application first calls `PackDDElParam()` and sends its return value (a pointer cast to `LPARAM`) as the `lParam` in `PostMessage()`.

Normally the receiving application is responsible for freeing the structure [via `FreeDDElParam()`]. However, if the call to `PostMessage()` fails, the posting application must free the packed data. This is also the method used by 16-bit Windows

Additional reference words: 3.10

INF: System Versus User Locale Identifiers

Article ID: Q100488

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

In Windows NT version 3.1, the following pairs of application programming interfaces (APIs) have the same functionality:

GetSystemDefaultLCID() and GetUserDefaultLCID()
GetSystemDefaultLangID() and GetUserDefaultLangID()

The user LangID and LCID are always set to the system value. In future versions of Windows NT, it will be possible to set the LangID and the LCID on a per-user basis.

Note that it is possible to set the LCID on a per-thread basis [that is, SetThreadLocale()] in Windows NT 3.1.

Additional reference words: 3.10

INF: Multiline Edit Control Limits in Windows NT
Article ID: Q89712

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The default maximum size for a multiline edit (MLE) control in both Windows and Windows NT is 30,000 characters. The EM_LIMITTEXT message allows an application to increase this value. Setting "cchmax" to 0 is a portable method of increasing this limit to the maximum in both 16-bit and 32-bit Windows. When cchmax is set to 0, the maximum size for an MLE is 4GB-1 (4 gigabytes minus 1).

Additional reference words: 3.10 3.1 win16 win32

INF: Use of DLGINCLUDE in Resource Files

Article ID: Q91697

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Windows 3.1 SDK dialog editor needs a way to know what include file is associated with a resource file that it opens. Rather than prompt the user for the name of the include file, the name of the include file is embedded in the resource file in most cases.

Embedding the name of the include file is done with a resource of type RCDATA with the special name DLGINCLUDE. This resource is placed into the .RES file and contains the name of the include file. The dialog editor looks for this resource when it loads a .RES file. If this resource is found, then the include file is opened also; if not, the editor prompts the user for the name of the include file.

In some Windows 3.1 build environments, the dialog editor was used to create dialogs that were placed in more than one .DLG file. These different .DLG files were then included in one .RC file, which was compiled with the resource compiler. Therefore, the resource file gets multiple copies of a RCDATA type resource with the same name, DLGINCLUDE, but the resource compiler and dialog editor don't complain.

In the Win32 SDK, changes were made so that this resource has its own resource type; it was changed from an RCDATA-type resource with the special name, DLGINCLUDE, to a DLGINCLUDE resource type whose name can be specified. The dialog editor would look for resources of the type DLGINCLUDE.

Changes were made to CvtRes so that it gives an error if it finds a resource that has the same type, name, and language as another resource in the file. We are being more strict about the need for resources to be unique in the Win32 SDK than the Windows 3.1 SDK. This is good because there was never any guarantee at run time as to which of the two or more resources would be returned by LoadResource().

This means that some applications being ported to Windows NT give an error when their resources are compiled because they have duplicate RCDATA type resources with the same name (DLGINCLUDE). This error is by design. The workaround is straightforward: delete all the DLGINCLUDE RCDATA type resource statements from all the .DLG files.

Finally, because it does not make sense to have the DLGINCLUDE type resources in the executable, CvtRes will strip them out so that they don't get linked into the EXE.

Additional reference words: 3.10 3.1

INF: Multiple Desktops Under Windows NT
Article ID: Q92505

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Win32 does not support sharing the same desktop between two monitors on the same machine. An independent software vendor (ISV) that wants this capability must write a display device driver that behaves similar to a normal driver with a single monitor, but actually controls two monitors.

Additional reference words: 3.10 3.1

INF: Clarification of COMMPROP MaxTxQueue Members

Article ID: Q94950

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In the COMMPROP structure, the dwMaxTxQueue and dwMaxRxQueue members state "a value of 0 means that this field is not used" in their descriptions.

This statement means that the provider does not restrict you to maximum Rx and Tx queue values, and these members [returned by GetCommProperties()] should not be used to determine the size of your transmit and receive buffers when calling SetupComm().

Based on the memory present in the system, NT's serial driver determines a default Rx queue size (currently 128 bytes on low memory systems and 4K on high memory systems). The current Rx and Tx queue sizes are located in the dwCurrentTxQueue and dwCurrentRxQueue members.

SetupComm() allows you to change these default queue sizes. However, you should not assume that the given serial driver will allocate any memory. The queue size allocated is stored in the dwCurrentRxQueue member of the COMMPROP structure. You may use this information to set the XonLim and XoffLim members of the device control block (DCB) structure.

The Microsoft-supplied serial driver attempts to allocate at least the amount requested for the RXQUEUE and, failing this, the request will also fail. The driver never attempts to allocate memory for the TXQUEUE.

Additional reference words: 3.10 3.1

INF: OpenComm() and Related Flags Obsolete Under Win32
Article ID: Q94990

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

OpenComm(), a Windows 3.1 application programming interface (API), is obsolete under Win32 and is not in the Win32 API. Also, the flags IE_BADID, IE_BAUDRATE, IE_BYTESIZE, IE_DEFAULT, IE_HARDWARE, IE_MEMORY, IE_NOPEN, and IE_OPEN are obsolete, but are still in the header files. However, these flags will be removed from the retail release.

These flags and the OpenComm() API do exist for Win16 applications running under Windows on Windows (WOW).

Under Win32, CreateFile() is used to create a handle to a communications resource (for example, COM1). The fdwShareMode parameter must be 0 (exclusive access), the fdwCreate parameter must be OPEN_EXISTING, and the hTemplate parameter must be NULL. Read, write, or read/write access can be specified, and the handle can be opened for overlapped I/O.

The standard I/O API ReadFile() and WriteFile() are used for communications I/O. The TTY sample program shipped with the Win32 Software Development Kit (SDK) demonstrates how to do serial I/O under Win32.

Additional reference words: 3.10 3.1

INF: Window Message Priorities

Article ID: Q96006

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

With Win32, messages have the same priorities as with Windows 3.1.

With normal usage of GetMessage() (passing zeros for all arguments except the LPMSG parameter) or PeekMessage(), any message on the application queue is processed before user input messages. And input messages are processed before WM_TIMER and WM_PAINT "messages."

For example, PostMessage() puts a message in the application queue. However, when the user moves the mouse or presses a key, these messages are placed on another queue (the system queue in Windows 3.1; a private, per-thread input queue in Win32).

GetMessage() and its siblings never look at the user input queue until the application queue is empty. Also, the WM_TIMER and WM_PAINT messages are not handled until there are no other messages (for the thread) to process. The WM_TIMER and WM_PAINT messages can be thought of as boolean toggles, because multiple WM_PAINT or WM_TIMER messages waiting in the queue will be combined into one message. This reduces the number of times a window must repaint itself.

Under this scheme, prioritization can be considered tri-level. All posted messages are higher priority than user input messages because they reside in different queues. And all user input messages are higher priority than WM_PAINT and WM_TIMER messages.

The only difference in the Win32 model from the Windows versions 3.x model is that there is effectively a system queue per thread (for user input messages) rather than one global system queue. The prioritization scheme for messages is identical.

More Information:

For information concerning SendMessage() from one thread to another, query on the following words in the Microsoft Knowledge Base:

Win32 and SendMessage() and overview

Additional reference words: 3.10

INF: Distinguishing Between Keyboard ENTER and Keypad ENTER
Article ID: Q96242

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

It is possible using `ReadConsoleInput()` or `PeekConsole()` to distinguish between the ENTER key on the main keyboard and the ENTER key on the numeric keypad. The `KEY_EVENT_RECORD` structure in the `INPUT_RECORD` structure must be used to distinguish between the two keys.

The following illustrates what the `KEY_EVENT_RECORD` structure is filled with after a keyboard ENTER key versus a numeric keypad ENTER key is pressed.

Keyboard ENTER Key

```
KeyEvent.wRepeatCount      = 1
KeyEvent.wVirtualKeyCode   = 13
KeyEvent.wVirtualScanCode  = 28
KeyEvent.dwControlKeyState = 00000000
```

Numeric Keypad ENTER Key

```
KeyEvent.wRepeatCount      = 1
KeyEvent.wVirtualKeyCode   = 13
KeyEvent.wVirtualScanCode  = 28
KeyEvent.dwControlKeyState = 00000100
```

In the case of the numeric keypad ENTER key, in `dwControlKeyState`, `ENHANCED_KEY` bit is set.

Additional reference words: 3.10

PRB: Setting Hooks Locally or Globally

Article ID: Q97919

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The application programming interface (API) SetWindowsHookEx() allows the developer to control whether hooks get set globally or locally. As with Windows version 3.1, the API SetWindowsHook() is used for local hooks only. The documentation does not make this distinction clear.

Additional reference words: 3.10

INF: NULL is a Valid Return From SetWindowsHook()

Article ID: Q97920

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

NULL may be returned as a valid handle from a call to SetWindowsHook(). NULL will be returned when installing the first hook of a particular hook type (with the exception of the HK_KEYBOARD and HK_MESSAGE hook types). This behavior is compatible with Windows version 3.1.

GetLastError() returns something useful only when an error is returned from an API. Because there is no way to tell if this API failed, GetLastError() will not return a meaningful value.

The SetWindowsHook() API returns NULL when you try to set a WH_MSGFILTER, WH_MOUSE, or WH_DEBUG hook with it. GetLastError() returns 0x57 "invalid parameter". However, the hook procedure does get called normally and the hook is successfully set.

Additional reference words: 3.10

INF: LB_GETCARETINDEX Returns 0 for Zero Entries in List Box
Article ID: Q97922

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

To determine whether a multiple selection list box is empty or has no items to select, two messages are required. First, call LB_GETCOUNT to determine whether or not the list box is empty. If the list box is not empty, then use LB_GETCARETINDEX to determine the position of the caret.

If you want a list box to contain selections that remain after the focus goes elsewhere, Microsoft recommends using visible check marks next to the items in the list box. This method provides better visual feedback to the user than a selection bar.

Additional reference words: 3.10 listbox

INF: SetActiveWindow() and SetForegroundWindow() Clarification
Article ID: Q97925

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

By default, each thread has an independent input state (its own active window, its own focus window, and so forth). SetActiveWindow() always logically sets a thread's active window state. To force a window to the foreground, however, use SetForegroundWindow(). SetForegroundWindow() activates a window and forces the window into the foreground. SetActiveWindow() always activates, but it brings the active window into the foreground only if the thread is the foreground thread.

Applications can call AttachThreadInput() to allow a set of threads to share the same input state. By sharing input state, the threads share their concept of the active window. By doing this, one thread can always activate another thread's window. This application programming interface (API) is also useful for sharing focus state, mouse capture state, keyboard state, and window Z-order state among windows created by different threads whose input state is shared.

Additional reference words: 3.10

INF: Possible Serial Baud Rates on Various Machines

Article ID: Q99026

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

Computers running Windows NT may be unable to set the same serial baud rates due to differences in serial port hardware on various platforms and machines. These differences may be important to note when writing a serial communications application that runs on different Windows NT platforms.

The simplest way to determine what baud rates are available on a particular serial port is to call the GetCommProperties() application programming interface (API) and examine the COMMPROP.dwSettableBaud bitmask to determine what baud rates are supported on that serial port.

More Information:

Some baud rates may be available on one machine and not on another because of differences in the serial port hardware used on the two machines. Most Intel 80x86 machines use a standard 1.8432 megahertz (MHz) clock speed on serial port hardware, and therefore most Intel machines can set the same baud rates. However, on other platforms, such as MIPS, there is no standard serial port clock speed. MIPS serial ports are known to exist with 1.8432 MHz, 3.072 MHz, 4.2336 MHz, and 8.0 MHz serial port clock chips. Future NT implementations on other platforms may have different serial port clock speeds as well.

Furthermore, certain requested baud rates are special-cased in the Windows NT serial driver so that they will work. The following are these special cases:

MHz	Requested Baud	Divisor	Resulting Baud Rate (+/- 1)
1.8432	56000	2	57600
3.072	14400	13	14769
4.2336	9600	28	9450
4.2336	14400	18	14700
4.2336	19200	14	18900
4.2336	38400	7	37800
4.2336	56000	5	52920
8.0	14400	35	14286
8.0	56000	9	55556

The actual baud rate can be calculated by dividing the divisor multiplied by 16 into the clock rate. For example, for a 1.8432 MHz clock and a divisor of 2, the baud rate would be:

$$1843200 \text{ Hz} / (2 * 16) = 57600$$

For all other cases, as long as the requested baud rate is within 1 percent of the nearest baud rate that can be found with an integer divisor, the baud rate request will succeed.

Additional reference words: 3.10 3.1

INF: Memory Handles and Icons

Article ID: Q99360

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

Memory handles are not globally sharable among processes. Handles for icons, cursors, windows, and so forth, are not global handles but are handles into an index into the server-side handle table (the handle is actually an index into the server-side's handle table). Thus, some objects can be shared between processes (but probably shouldn't be, for concurrency reasons).

GDI-related objects, however, are stored in a client-side handle table, which is translated to a handle value in a server-side table on every client-server transition. Thus, there are some objects that can be shared (USER-related objects) and some that can't be shared (BASE/KERNEL and GDI).

There are three types of handles in the system:

- Handles to objects that the executive (object manager) knows about. These are assigned on a per-process basis but each access to these objects goes through the executive.
- Handles that are maintained by the Win32 subsystem server (USER objects, including icons and cursors) and are therefore sharable. Please note that the allowed behavior of shared USER objects is subject to change in future releases of Windows NT. Thus, care should be taken when using these handles.
- Handles that are maintained by the Win32 subsystem client, and therefore are valid only in the context of the process that created it (GDI objects). These handles differ from the first type of handles listed in that you cannot call handle manipulation functions, such as DuplicateHandle() or WaitForSingleObject(), or use the security facilities on these objects.

Additional reference words: 3.10

INF: Debugging a System-Wide Hook

Article ID: Q102428

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Debugging a system-wide hook such as a journal hook must be done with the extreme caution. When an application installs such a hook, it effectively takes control of user input. In effect, this disables the interface with the debugger. For example, after installing a journal record hook, you must unhook the record hook when you want to allow the debugger to regain control.

It is not possible to use an interactive debugger to debug an actively installed journal hook using a single machine. It is possible to use a remote debugger, because one interface can be blocked (or recording) while the other one does the debugging.

More Information:

System-wide input hook procedures can be thought of as being in three possible states:

- unhooked (not installed)
- suspended
- hooked (installed)

In the unhooked state, the procedure imposes no control over user input. In the hooked state, all user input specifically defined to be handled by this hook passes through this procedure. In the suspended state, all user input specifically defined to be handled by this hook is completely blocked.

In the case of a journal record hook, the suspended state can be achieved when a breakpoint is reached within the hook procedure. When this happens, all user input (system wide, that is) in the form of mouse and keyboard input is blocked, and thus you cannot interact with the debugger or any other application as you normally would. Fortunately, when the user presses the CTRL+ESC or the CTRL+ALT+DEL key combinations, all system-wide hooks are automatically unhooked, returning the system to the unhooked state.

Once this has occurred, it is likely that the application with the journal hook is now in a undefined state (because it had the hook pulled out from underneath it, so to speak). Fortunately, the system will send all applications the WM_CANCELJOURNAL message to indicate that it has removed the hook. A well behaved application can intercept this message and adjust its state accordingly.

Additional reference words: 3.10

INF: WM_ENTERIDLE Documentation Is Misleading
Article ID: Q102446

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The Win32 application programming interface (API) online documentation for the WM_ENTERIDLE message states the following:

The WM_ENTERIDLE message informs an application's main window procedure that a modal dialog box or menu is entering an idle state.

This is incorrect for dialog boxes. The WM_ENTERIDLE message is sent to the dialog box's owner window rather than the application's main window as the documentation states.

Microsoft has confirmed this to be a problem in the Windows NT SDK version 3.1.

A modal dialog box or menu enters an idle state when no messages are waiting in its queue after it has processed one or more previous messages.

Additional reference words: 3.10

INF: How to Make SPINCUBE a Global Class

Article ID: Q102483

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The spincube class (from the SDK sample SPINCUBE) in SPINCUBE.DLL can be made a global class that is initialized at logon time. To do this, put the dynamic-link library (DLL) name (full path) in the registry (the DLL registers its classes via DLL_PROCESS_ATTACH) under the section:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\Windows  
APPINIT_DLLS
```

Once you add this key and log on again, any application in the system can use this global class.

Additional reference words: 3.10

INF: The SBS_SIZEBOX Style

Article ID: Q102485

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

A size box is a small rectangle the user can expand to change the size of the window. When you want a size box, you create a SCROLLBAR window with the SBS_SIZEBOX flag. This action creates a size box with the height, width, and position that you specified in the call to CreateWindow. If you specify SBS_SIZEBOXBOTTOMRIGHTALIGN, the box will be aligned in the lower right of the rectangle you specified when creating the window. If you specify SBS_SIZEBOXTOPLEFTALIGN, the box will be aligned in the upper left of the rectangle you specified in your call to CreateWindow().

MORE INFORMATION

=====

The user moves the mouse pointer over to the box, presses and holds the left mouse button, and drags the mouse pointer to resize the window. When the user does this, the borders on the window (the frame) move. When the user releases the mouse button, the window is resized.

You create a size box by creating a child window of type WS_CHILD | WS_VISIBLE | SBS_SIZEBOX | SBS_SIZEBOXTOPLEFTALIGN. You don't have to do any of the processing for this; the system will take care of it. You will notice in your window procedure that you will get the scrollbar messages plus the WM_MINMAXINFO message. Size boxes work similar to the way the WS_THICKFRAME/WS_SIZEBOX style does on a window.

Additional reference words: sizebox scrollbar

SAMPLE: Control Panel Application Sample

Article ID: Q102486

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

A sample demonstrating the creation of a Control Panel application is now available.

The CPLApplet function is for a dynamic-link library (DLL) containing three Control Panel applications that set preferences for a component stereo system attached to the computer. The sample uses an application-defined StereoApplets array that contains three structures, each corresponding to one of the Control Panel applications. Each structure contains all the information required by the CPL_NEWINQUIRE message, as well as the dialog box template and dialog box procedure required by the CPL_DBLCLK message. The code demonstrates how to fill the structures in the StereoApplets array.

To install your new DLL, copy the .CPL file into your SYSTEM32 directory and the Control Panel will automatically load the DLL upon startup.

Additional reference words: 3.10

**INF: Clarification of the
Article ID: Q102765**

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Under Windows NT, the "Country" choice affects currency, date/time and number format, and so forth. The "Language" choice affects sorting, names of the days of the weeks and months, and so forth. These settings allow the user to choose the appropriate language and country format. For example, if you are British and living in the U.S., you can pick a locale of English (British) at setup time, then use Control Panel later to change your country to U.S. so that currency is in dollars instead of pounds.

Additional reference words: 3.10

PRB: CloseClipboard() Suggests Calling DuplicateHandle()
Article ID: Q103240

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SYMPTOMS

=====

The documentation for CloseClipboard() suggests using DuplicateHandle() on a clipboard object before closing the clipboard, in order to use that object after the clipboard is closed. Doing so results in a return of ERROR_INVALID_HANDLE.

CAUSE

=====

The documentation is in error. DuplicateHandle() is specified to work only on console input, console output, events, files, file mappings, mutexes, pipes, processes, semaphores, and thread handles.

Additional reference words: 3.10

INF: Differences Between hInstance on Win 3.1 and Windows NT
Article ID: Q103644

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

In Microsoft Windows version 3.1, an instance handle can be used to uniquely identify the instance of an application because instance handles are unique within the scope of an address space. Because each instance of an application runs in its own address space on Windows NT, instance handles cannot be used to uniquely identify an instance of an application running on the system. This article explains why, and some alternative calls that might assist in uniquely identifying an application instance on Windows NT.

MORE INFORMATION

=====

Although the concepts for an instance handle are similar on Windows NT and Windows 3.1, the results you see regarding them might be very different from what you expect.

With Windows 3.1, when you start several instances of the same application, they all share the same address space. You have multiple instances of the same code segment; however, each of these instances has a unique data segment associated with it. Using an instance handle (hInstance) is a way to uniquely identify these different instances and data segments in the address space.

Instance handles are unique to the address space. On Windows NT, when looking at the value of the instance handle, or the value returned from `GetWindowLong(hWnd, GWL_HINSTANCE)`, a developer coming from a Windows 3.1 background might be surprised to see that most of the windows on the desktop return the same value. This is because the return value is the hInstance for the instance of the application, which is running in its own address space. (An interesting side note: The hInstance value is the base address where the application's module was able to load; either the default address or the fixed up address.)

In Win32 on Windows NT, running several instances of the same application causes the instances to start and run in their own separate address space. To emphasize the difference: multiple instances of the same application on Windows 3.1 run in the same address space; in Windows NT, each instance has its own, separate address space. Using an instance handle to uniquely identify an application instance, as is possible on Windows 3.1, does not apply in Windows NT. (Another interesting side note: Remember that even if there are multiple instances of an application, if they are able to load at their default virtual address spaces, the virtual address

pages of the different applications' executable code will map to the same physical memory pages.)

In Windows NT, instance handles are not unique in the global scope of the system; however, window handles, thread IDs, and process IDs are. Here are some calls that may assist in alternative methods to uniquely identify instance of applications on Windows NT:

- `GetWindowThreadProcessID()` retrieves the identifier of the thread that created the given window and, optionally, the identifier of the process that created the window.
- `OpenProcess()` returns a handle to a process specified by a process ID.
- `GetCurrentProcessID()` returns the calling process's ID.
- `EnumThreadWindows()` returns all of the windows associated with a thread.
- The `FindWindow()` function retrieves the handle of the top-level window specified by class name and window name.
- To enumerate all of the processes on the system, you can query the Registry using `RegQueryValueEx()` with key `HKEY_PERFORMANCE_DATA`, and the Registry database index associated with the database string "Process".

For further details on using these calls, please see the Win32 SDK Help files.

Additional reference words: 3.00 3.10

INF: Propagating Environment Variables to the System
Article ID: Q104011

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

User environment variables can be modified using the System application or by editing the following Registry key:

```
HKEY_CURRENT_USER \
    Environment
```

System environment variables can be modified by editing the following Registry key:

```
HKEY_LOCAL_MACHINE \
    SYSTEM \
    CurrentControlSet \
        Control \
            Session Manager \
                Environment
```

Note, however, that modifications to the environment variables do not result in immediate change. For example, if you start another Command Prompt after making the changes, the environment variables will reflect the previous (not the current) values. The changes do not take effect until you log off and then log back on.

To effect these changes without having to log off, broadcast a WM_WININICHANGE message to all windows in the system, so that any interested applications (such as Program Manager, Task Manager, Control Panel, and so forth) can perform an update.

MORE INFORMATION

=====

For example, the following code fragment should propagate the changes to the environment variables used in the Command Prompt:

```
SendMessage( FindWindow( "Progman", NULL ), WM_WININICHANGE,
    0L, (LPARAM) "Environment" );
```

Additional reference words: 3.10

INF: 32-Bit Scroll Ranges

Article ID: Q104311

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

You can use 32-bit scroll ranges by calling `GetScrollPos()`; however, you cannot get 32-bit positions for notifications sent while thumb tracking, that is, via the `SB_THUMBPOSITION` message. This is because thumb position information is not queryable via an application programming interface (API). You only can obtain the 32-bit scroll information only before or after the scroll has taken place.

The scroll bar APIs allow setting a scroll range up to `0x7FFFFFFF` via `SetScrollRange()`, and setting a scroll position within that range using `SetScrollPos()`. If the `WM_HSCROLL` or `WM_VSCROLL` message is processed, the information returned for scroll bar position, `nPos`, is only a 16-bit value. To obtain the 32-bit information, the `GetScrollPos()` API must be used.

Additional reference words: 3.10 scrollbar

INF: COMCTL32 APIs Unsupported in the Win32 SDK
Article ID: Q105300

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The library COMCTL32.LIB was included in the Win32 SDK because the PERFMON sample makes use of it. The library is part of the Windows for Workgroups (WFW) COMMCTRL.LIB. However, Microsoft does not officially support COMCTL32.LIB or recommend the use of these application programming interfaces (APIs), and therefore they have not been documented in the SDK.

Microsoft does not recommend these APIs because most of the interfaces will change in future versions of Windows and Windows NT. Therefore, Microsoft will not be supporting COMCTL32.

If you must absolutely use COMCTL32 at this time, the documentation can be found in the WFW SDK. Be aware that the code that you write will break in future operating systems.

Additional reference words: 3.10

INF: Cancelling WaitCommEvent() with SetCommMask()

Article ID: Q105302

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

If a serial port is in nonoverlapped mode (without `FILE_FLAG_OVERLAPPED`) and `SetCommMask()` is called, the call does not return until any pending `WaitCommEvent()` calls return. This apparently contradicts the following statement from the `SetCommMask()` Help

If `SetCommMask()` is called for a communications resource while a wait is pending for that resource, `WaitCommEvent()` returns an error.

and the following statement from the `WaitCommEvent()` Help:

If a process attempts to change the device handle's event mask by using the `SetCommMask()` function while a `WaitCommEvent()` operation is in progress, `WaitCommEvent()` returns immediately.

However, this is the expected behavior. If you open a serial port in the nonoverlapped mode, then you can do only one thing at a time with the serial port. `SetCommMask()` must block while the `WaitCommEvent()` call is blocking.

If the serial port was opened with `FILE_FLAG_OVERLAPPED`, `WaitCommEvent()` will return after `SetCommEvent()` has been called.

Additional reference words: 3.10 com1 com2

INF: Win32 Shell Dynamic Data Exchange (DDE) Interface
Article ID: Q105446

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

Information on the DDE Interface to Program Manager can be found in the Win32 application programming interface (API) reference under the topic "Shell Dynamic Data Exchange Interface."

The SDK contains a sample program that interfaces with Program Manager. The sample can be found in MSTOOLS\SAMPLES\DDEML\DDEPROG.

MORE INFORMATION

=====

AppProperties cannot be used to get the item icon, description, or working directory, as it can in Windows 3.1. Therefore, GetIcon(), GetDescription(), and GetWorkingDir() do not work in Windows NT. However, AppProperties can still be used to dump out the contents of a group, by specifying the group name in lParam.

Here's how a Win32 application can get the item icon, the description, and the working directory:

1. Initiate a conversation with the Shell as follows

```
SendMessage( -1, WM_DDE_INITIATE, hWndApp, lParam );
```

where lParam points to an atom representing:

```
LOWORD | HIWORD
```

```
Shell | AppIcon           : To get an item's icon  
Shell | AppDescription   : To get an item's description  
Shell | AppWorkingDir    : To get an item's working directory
```

2. Get the item DDE number.

The DDE number is stored by Program Manager in the STARUPINFO structure of the application when the application is started. The application can get the startup information with:

```
GetStartupInfo( &StartupInfo );
```

The field lpReserved in the STARUPINFO structure is in the following format

```
dde.#, hotkey.##
```

where the DDE number is # and the hot key for the item is ##.

3. Request data as follows

```
SendMessage( hwndProgMan, WM_DDE_REQUEST, hwndApp, lParam );
```

where the lParam HIWORD is the item's DDE number obtained in step 2.

4. The data is returned in lParam of WM_DDE_DATA message. The DDE data value is a string for AppDescription and AppWorkingDir DDE transactions. For AppIcon, the data value has the following structure:

```
typedef struct _PMIconData {  
    DWORD dwResSize;  
    DWORD dwVer;  
    BYTE iResource; // icon resource  
} PMICONDATA, *LPPMICONDATA;
```

To create the icon, the application must call:

```
hIcon = CreateIconFromResource((LPBYTE)&(lpPMIconData->iResource),  
    lpPMIconData->dwResSize,  
    TRUE,  
    lpPMIconData->dwVer  
    );
```

Additional reference words: 3.10

INF: Win32 Drag and Drop Server

Article ID: Q105530

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The supported method for creating a drag and drop server is to use OLE (Object Linking and Embedding) version 2.0. This works on Windows and Windows NT and will work on future versions of these operating systems.

Additional reference words: 3.10

INF: ClipCursor() Requires WINSTA_WRITEATTRIBUTES
Article ID: Q106384

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The documentation for ClipCursor() states that the calling process must have WINSTA_WRITEATTRIBUTES access to the window station. However, this permission does not have to be enabled programmatically because the system always grants the Local Login SID (the interactive user) WINSTA_WRITEATTRIBUTES access, regardless of the permission groups to which the user belongs.

MORE INFORMATION

=====

The following code shows how to confine the cursor to the application window during WM_ACTIVATE processing. Note that the clip cursor region must be restored to its previous state each time the application deactivates.

Sample Code

```
static RECT rcOldClip;
.
.
.

case WM_ACTIVATE:{
    short fActive = LOWORD( wParam );

    if( fActive ){
        RECT rcNewClip;

        /* Record the area in which the cursor can move. */
        GetClipCursor( &rcOldClip );

        /* Get the dimensions of the application's client area. */
        GetClientRect( hWnd, &rcNewClip);

        /* Convert to screen coordinates. */
        MapWindowPoints( hWnd, NULL, (LPPOINT)&rcNewClip, 2 );

        /* Confine the cursor to the application's window. */
        ClipCursor( &rcNewClip );
    }
    else{
        /* Restore the cursor to its previous area. */
```



```
        ClipCursor( &rcOldClip );  
    }  
    break;  
}
```

Additional reference words: 3.10

INF: Retrieving DIBs from the Clipboard

Article ID: Q106386

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

Retrieving a DIB (device-independent bitmap) from the clipboard can take significantly more time than retrieving a bitmap from the clipboard. The difference stems from the fact that a bitmap is a GDI object and a DIB is a global memory object.

When SetClipboardData() is passed a global memory handle, as it is when it is passed a handle to a DIB, all the data gets copied into the Win32 server and put into a sharable section of memory. When the DIB is retrieved with GetClipboardData(), the shared memory is mapped into the application's virtual address space and the memory handle is cached. Any subsequent calls to GetClipboardData() return quickly, because the memory does not have to be remapped.

In contrast, when retrieving a bitmap with GetClipboardData(), only a handle is created, because a bitmap is a GDI object.

When CloseClipboard() is called, all of the cached handles to shared memory and GDI objects are deleted.

Rather than reopening the clipboard, it is a good idea to keep a local copy of anything retrieved from the clipboard if the item will be used again after the clipboard has been closed. In general, data should be retrieved from the clipboard only when the application is doing a paste or if the application is a clipboard viewer processing a WM_DRAWCLIPBOARD message.

MORE INFORMATION

=====

The data for a GDI object exists on the server side. In other words, bitmaps and DDBs (device-dependent bitmaps) exist in the Win32 subsystem address space. Only the handles of GDI objects are private to an application. Therefore, to make a bitmap or a DDB accessible to another application, only a call to DuplicateHandle() is needed.

Note that even though it is faster to retrieve a DDB from the clipboard, it is still recommended to put a DIB on the clipboard rather than a DDB.

Additional reference words: 3.10

INF: NEW.H Does Not Contain new() that Takes a void*
Article ID: Q99871

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The form of operator new that takes a parameter of type "void *" was added in the March '93 ANSI C++ draft document, as Section 17.1.1.3. As such, it is not supported by the Win32 Software Development Kit (SDK) compiler.

This feature may be added to a future Microsoft C/C++ compiler; however, there are certain considerations. If this addition is made to NEW.H, then every C++ program that includes NEW.H in more than one module will receive multiple definition errors. Either the definition must be declared in the CRT or declared INLINE. Furthermore, because the user should be able to replace the definition of "operator new", the definition must be replaced in the CRT to ensure that it is not inlined into some modules, which would prevent it from being user-replaced.

Additional reference words: 3.10

INF: Win32 Equivalents for C Run-Time Functions

Article ID: Q99456

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

Many of the C run-time functions have direct equivalents in the Win32 application programming interface (API). This article lists the C run-time functions by category with their Win32 equivalents or the word "none" if no equivalent exists.

Note that the functions that are followed by an asterisk (*) are part of the 16-bit C run-time only. Functions that are unique to the 32-bit C run time are listed separately in the last section. All other functions are common to both C run times.

Buffer Manipulation

_memccpy	none
memchr	none
memcmp	none
memcpy	CopyMemory
_memicmp	none
memmove	MoveMemory
memset	FillMemory, ZeroMemory
_swab	none

Character Classification

isalnum	IsCharAlphaNumeric
isalpha	IsCharAlpha, GetStringTypeW (Unicode)
__isascii	none
__iscntrl	none, GetStringTypeW (Unicode)
__iscsym	none
__iscsymf	none
isdigit	none, GetStringTypeW (Unicode)
isgraph	none
islower	IsCharLower, GetStringTypeW (Unicode)
isprint	none
ispunct	none, GetStringTypeW (Unicode)
isspace	none, GetStringTypeW (Unicode)
isupper	IsCharUpper, GetStringTypeW (Unicode)
isxdigit	none, GetStringTypeW (Unicode)
__toascii	none
tolower	CharLower
_tolower	none
toupper	CharUpper
_toupper	none

Directory Control

```

-----
_chdir          SetCurrentDirectory
_chdrive        SetCurrentDirectory
_getcwd         GetCurrentDirectory
_getdrive      GetCurrentDirectory
_mkdir         CreateDirectory
_rmdir         RemoveDirectory
_searchenv     SearchPath

```

File Handling

```

-----
_access        none
_chmod         SetFileAttributes
_chsize        SetEndOfFile
_filelength    GetFileSize
_fstat         See Note 5
_fullpath      GetFullPathName
_get_osfhandle none
_isatty        GetFileType
_locking       LockFileEx
_makepath      none
_mktemp        GetTempFileName
_open_osfhandle none
_remove        DeleteFile
_rename        MoveFile
_setmode       none
_splitpath     none
_stat          none
_umask         none
_unlink        DeleteFile

```

Creating Text Output Routines

```

-----
_displaycursor* SetConsoleCursorInfo
_gettextcolor*  GetConsoleScreenBufferInfo
_gettextcursor* GetConsoleCursorInfo
_gettextposition* GetConsoleScreenBufferInfo
_gettextwindow* GetConsoleWindowInfo
_outtext*       WriteConsole
_scrolltextwindow* ScrollConsoleScreenBuffer
_settextcolor*  SetConsoleTextAttribute
_settextcursor* SetConsoleCursorInfo
_settextposition* SetConsoleCursorPosition
_settextwindow* SetConsoleWindowInfo
_wrapon*        SetConsoleMode

```

Stream Routines

```

-----
clearerr       none
fclose         CloseHandle
_fclosetall    none
_fdopen        none
feof           none
ferror         none
fflush         FlushFileBuffers
fgetc          none
fgetchar      none

```

fgetpos	none
fgets	none
_fileno	none
_flushall	none
fopen	CreateFile
fprintf	none
fputc	none
_fputchar	none
fputs	none
fread	ReadFile
freopen (std handles)	SetStdHandle
fscanf	none
fseek	SetFilePointer
fsetpos	SetFilePointer
_fsopen	CreateFile
ftell	SetFilePointer (check return value)
fwrite	WriteFile
getc	none
getchar	none
gets	none
_getw	none
printf	none
putc	none
putchar	none
puts	none
_putw	none
rewind	SetFilePointer
_rmtmp	none
scanf	none
setbuf	none
setvbuf	none
_snprintf	none
sprintf	wsprintf
sscanf	none
_tempnam	GetTempFileName
tmpfile	none
tmpnam	GetTempFileName
ungetc	none
vfprintf	none
vprintf	none
_vsnprintf	none
vsprintf	wvsprintf

Low-Level I/O

_close	_lclose, CloseHandle
_commit	FlushFileBuffers
_creat	_lcreat, CreateFile
_dup	DuplicateHandle
_dup2	none
_eof	none
_lseek	_llseek, SetFilePointer
_open	_lopen, CreateFile
_read	_lread, ReadFile
_sopen	CreateFile
_tell	SetFilePointer (check return value)
_write	_lread

Console and Port I/O Routines

_cgets	none
_cprintf	none
_cputs	none
_cscanf	none
_getch	ReadConsoleInput
_getche	ReadConsoleInput
_inp	none
_inpw	none
_kbhit	PeekConsoleInput
_outp	none
_outpw	none
_putch	WriteConsoleInput
_ungetch	none

Memory Allocation

_alloca	none
_bfreeseg*	none
_bheapseg*	none
_calloc	GlobalAlloc
_expand	none
_free	GlobalFree
_freect*	GlobalMemoryStatus
_halloc*	GlobalAlloc
_heapadd	none
_heapchk	none
_heapmin	none
_heapset	none
_heapwalk	none
_hfree*	GlobalFree
_malloc	GlobalAlloc
_memavl	GlobalMemoryStatus
_memmax	GlobalMemoryStatus
_msize*	GlobalSize
_realloc	GlobalReAlloc
_set_new_handler	none
_set_hnew_handler*	none
_stackavail*	none

Process and Environment Control Routines

abort	none
assert	none
atexit	none
_cexit	none
_c_exit	none
_exec functions	none
_exit	ExitProcess
_exit	ExitProcess
_getenv	GetEnvironmentVariable
_getpid	GetCurrentProcessId
_longjmp	none
_onexit	none
_perror	FormatMessage

_putenv	SetEnvironmentVariable
_raise	RaiseException
_setjmp	none
_signal (ctrl-c only)	SetConsoleCtrlHandler
_spawn functions	CreateProcess
_system	CreateProcess

String Manipulation

strcat, wscat	lstrcat
strchr, wcschr	none
strcmp, wcscmp	lstrcmp
strcpy, wcsncpy	lstrcpy
strcspn, wcscspn	none
_strdup, _wcsdup	none
_strerror	FormatMessage
_strerror	FormatMessage
_stricmp, _wcsicmp	lstrcmpi
_strlen, wcslen	lstrlen
_strlwr, _wcslwr	CharLower, CharLowerBuffer
_strncat, wcsncat	none
_strncmp, wcsncmp	none
_strncpy, wcsncpy	none
_strnicmp, _wcsnicmp	none
_strnset, _wcsnset	FillMemory, ZeroMemory
_strpbrk, wcpbrk	none
_strrchr, wcsrchr	none
_strrev, _wcsrev	none
_strset, _wcsset	FillMemory, ZeroMemory
_strspn, wcsspn	none
_strstr, wcsstr	none
_strtok, wcstok	none
_strupr, _wcsupr	CharUpper, CharUpperBuffer

MS-DOS Interface

_bdos*	none
_chain_intr*	none
_disable*	none
_dos_allocmem*	GlobalAlloc
_dos_close*	CloseHandle
_dos_commit*	FlushFileBuffers
_dos_creat*	CreateFile
_dos_creatnew*	CreateFile
_dos_findfirst*	FindFirstFile
_dos_findnext*	FindNextFile
_dos_freemem*	GlobalFree
_dos_getdate*	GetSystemTime
_dos_getdiskfree*	GetDiskFreeSpace
_dos_getdrive*	GetCurrentDirectory
_dos_getfileattr*	GetFileAttributes
_dos_getftime*	GetFileTime
_dos_gettime*	GetSystemTime
_dos_getvect*	none
_dos_keep*	none
_dos_open*	OpenFile
_dos_read*	ReadFile

<code>_dos_setblock*</code>	GlobalReAlloc
<code>_dos_setdate*</code>	SetSystemTime
<code>_dos_setdrive*</code>	SetCurrentDirectory
<code>_dos_setfileattr*</code>	SetFileAttributes
<code>_dos_setftime*</code>	SetFileTime
<code>_dos_settime*</code>	SetSystemTime
<code>_dos_setvect*</code>	none
<code>_dos_write*</code>	WriteFile
<code>_dosexterr*</code>	GetLastError
<code>_enable*</code>	none
<code>_FP_OFF*</code>	none
<code>_FP_SEG*</code>	none
<code>_harderr*</code>	See Note 1
<code>_hardresume*</code>	See Note 1
<code>_hardretn*</code>	See Note 1
<code>_int86*</code>	none
<code>_int86x*</code>	none
<code>_intdos*</code>	none
<code>_intdosx*</code>	none
<code>_segread*</code>	none

Time

<code>asctime</code>	See Note 2
<code>clock</code>	See Note 2
<code>ctime</code>	See Note 2
<code>difftime</code>	See Note 2
<code>_ftime</code>	See Note 2
<code>_getsystime</code>	GetDateAndTime
<code>gmtime</code>	See Note 2
<code>localtime</code>	See Note 2
<code>mktime</code>	See Note 2
<code>_strdate</code>	See Note 2
<code>_strtime</code>	See Note 2
<code>time</code>	See Note 2
<code>_tzset</code>	See Note 2
<code>_utime</code>	SetDateAndTimeFile

Virtual Memory Allocation

<code>_vfree*</code>	See Note 3
<code>_vheapinit*</code>	See Note 3
<code>_vheapterm*</code>	See Note 3
<code>_vload*</code>	See Note 3
<code>_vlock*</code>	See Note 3
<code>_vlockcnt*</code>	See Note 3
<code>_vmalloc*</code>	See Note 3
<code>_vmsize*</code>	See Note 3
<code>_vrealloc*</code>	See Note 3
<code>_vunlock*</code>	See Note 3

32-Bit C Run Time

<code>_beginthread</code>	CreateThread
<code>_cwait</code>	WaitForSingleObject w/ GetExitCodeProcess
<code>_endthread</code>	ExitThread
<code>_findclose</code>	FindClose

<code>_findfirst</code>	<code>FindFirstFile</code>
<code>_findnext</code>	<code>FindNextFile</code>
<code>_ftime</code>	<code>SetFileTime</code>
<code>_get_osfhandle</code>	<code>none</code>
<code>_open_osfhandle</code>	<code>none</code>
<code>_pclose</code>	<code>See Note 4</code>
<code>_pipe</code>	<code>CreatePipe</code>
<code>_popen</code>	<code>See Note 4</code>

Note 1: The `_harderr` functions do not exist in the Win32 API. However, much of their functionality is available through structured exception handling.

Note 2: The time functions are based on a format that is not used in Win32. There are specific Win32 time functions that are documented in the Help file.

Note 3: The virtual memory functions listed in this document are specific to the MS-DOS environment and were written to access memory beyond the 640K of RAM available in MS-DOS. Because this limitation does not exist in Win32, the standard memory allocation functions should be used.

Note 4: While `_pclose()` and `_popen()` do not have direct Win32 equivalents, you can (with some work) simulate them with the following calls:

<code>_popen</code>	<code>CreatePipe</code> <code>CreateProcess</code>
<code>_pclose</code>	<code>WaitForSingleObject</code> <code>CloseHandle</code>

Note 5: `GetFileInformationByHandle()` is the Win32 equivalent for the `_fstat()` C run-time function. However, `GetFileInformationByHandle()` is not supported by Win32s. `GetFileSize()`, `GetFileAttributes()`, `GetFileTime()`, and `GetFileTitle` are supported by Win32s.

Additional reference words: 3.10

Sample: Writing NTSD Extensions

Article ID: Q85885

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

This article and the associated sample (called NTSD) demonstrate how to write an NTSD extension for the NTSD debugger.

While debugging, it is often necessary to look up certain fields of a particular structure in the program. This process usually involves dumping the address of the structure in question and locating the specific fields in the dump, which can be tedious and inefficient.

With NTSD, programmers can write a dumping routine to be called by the NTSD debugger.

The routine must be in a DLL and have the following prototype:

```
void Routine (HANDLE, HANDLE, HANDLE, PNTSD_EXTENSION_APIS, LPSTR);
```

See the file DEBUG.C for details.

Then, to invoke the routine in NTSD, the user would do the following:

```
!module.routine argument
```

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

INF: Using a Mouse with MEP Under Windows NT
Article ID: Q83300

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Microsoft Editor (MEP) included with the Win32 Software Development Kit (SDK) can be used with a mouse to position the cursor.

By default, the mouse is not enabled for MEP. You must add the switch "usemouse:yes" (without the quotation marks) to the TOOLS.INI file under the [m mep] section. The TOOLS.INI file is a text file editable by MEP or Notepad.

To change the position of the cursor, first position the mouse pointer on the new location and then click the left mouse button.

INF: Macros to Facilitate Porting Applications to Windows NT
Article ID: Q83359

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following is a list of macros that make the porting of applications to Microsoft Windows NT easier.

More Information:

// NT Macros

```
#if defined(WIN32)

#define _fmemcpy(x,y,z)    memcpy(x,y,z)
#define _fmemcmp(x,y,z)   memcmp(x,y,z)
#define _fmemset(x,y,z)   memset(x,y,z)
#define _fmemicmp(x,y,z)  memicmp(x,y,z)
#define _fmemmove(x,y,z)  memmove(x,y,z)

#define _fstrcpy(x,y)     strcpy(x,y)
#define _fstrcmp(x,y)     strcmp(x,y)
#define _fstrcat(x,y)     strcat(x,y)
#define _fstrlen(x)       strlen(x)
#define _fstricmp(x,y)    stricmp(x,y)
#define _fstrstr(x,y)     strstr(x,y)
#define _fstrncpy(x,y,z)  strncpy(x,y,z)
#define _fstrncmp(x,y,z)  strncmp(x,y,z)
#define _fstrupr(x)      strupr(x)
#define _fstrlwr(x)       strlwr(x)
#define _fstrchr(x,y)     strchr(x,y)
#define _fstrrchr(x,y)    strrchr(x,y)
#define _fstrnicmp(x,y,z) strnicmp(x,y,z)
#define _fstrpbrk(x,y)    strpbrk(x,y)

#define _nfree(x)         free(x)
#define _nmalloc(x)       malloc(x)

#define _loadds           // not valid under NT

#define NT_GetWndInstance(hwnd) (HINSTANCE)GetWindowLong(hwnd, GWL_HINSTANCE)

#define NT_GetWndID(hwnd)      (UINT)GetWindowLong(hwnd, GWL_ID);

#define NT_ParseWM_COMMAND(id, ntfy, hwnd, wPar, lPar) \
    (id = LOWORD(wPar), ntfy = HIWORD(wPar), hwnd = (HWND)lPar)

#define NT_PostWM_COMMAND(hwnd, id, ntfy, hwndChild) \
```

```
        PostMessage(hwnd, WM_COMMAND, (UINT)MAKELONG(id, ntfy), (LONG)hwndChild)

#define NT_SendWM_COMMAND(hwnd, id, ntfy, hwndChild) \
    SendMessage(hwnd, WM_COMMAND, (UINT)MAKELONG(id, ntfy), (LONG)hwndChild)

#if !defined(LONG2POINT)
#define LONG2POINT(l, pt) ((pt).x=(SHORT)LOWORD(l), (pt).y=(SHORT)HIWORD(l))
#endif

// NT Equivalents for Windows

#else
#define NT_GetWndInstance(hwnd) (HINSTANCE)GetWindowWord(hwnd, GWW_HINSTANCE)

#define NT_GetWndID(hwnd) (UINT)GetWindowWord(hwnd, GWW_ID);

#define NT_ParseWM_COMMAND(id, ntfy, hwnd, wPar, lPar) \
    (id = wPar, ntfy = HIWORD(lPar), hwnd = (HWND)LOWORD(lPar))

#define NT_PostWM_COMMAND(hwnd, id, ntfy, hwndChild) \
    PostMessage(hwnd, WM_COMMAND, (UINT)id, MAKELONG(hwndChild, ntfy))

#define NT_SendWM_COMMAND(hwnd, id, ntfy, hwndChild) \
    SendMessage(hwnd, WM_COMMAND, (UINT)id, MAKELONG(hwndChild, ntfy))

#endif
```

INF: Correct Use of Try/Finally

Article ID: Q83670

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Try/finally, used correctly, helps to provide a robust application. However, if used incorrectly it can cause unnecessary overhead. any flow of control out of the try body of try/finally is an abnormal termination that can cause hundreds of instructions to be executed on an x86 system, and thousands on a MIPS machine, even if control leaves the try body via a control statement on the very last statement of the try body. The language definition states that control must leave the try body sequentially for normal termination to occur (that is, execution falls through the bottom of the try body).

The following sample demonstrates an incorrect use of try/finally:

```
/* Incorrect use of try/finally */
```

```
VOID  
function (  
    DWORD ... P1,  
    .  
    DWORD ... Pn  
)  
{  
  
    try {  
        if (...) {  
            .  
            .  
            return;  
        }  
        .  
        .  
    } finally {  
        .  
        .  
    }  
    return;  
}
```

The overhead can be avoided in the above example by moving the return AFTER the end of the finally clause. The following provides more detail on the correct use of try/finally.

More Information:

Execution of a termination handler due to abnormal termination of a try body is expensive. Abnormal termination occurs when control leaves a try body in any way other than by falling through the bottom. Intentionally branching out of a try body is still an abnormal termination.

In the above example, abnormal termination of the try body occurs if the return in the middle of the try body is executed. If the predicate of the if is false, then extremely efficient execution of the finally clause occurs because this is not abnormal termination and the finally clause is called directly by inline code.

When abnormal termination occurs hundreds to thousands of instructions are executed because an unwind must be executed, which must search backward through frames to determine if any termination handlers should be called. On an x86 system, this executes the C run-time handler and examines the handler list. On a MIPS machine, this also causes the function table to be searched and the prologue of each intervening function to be executed backwards interpretively.

You should always avoid the execution of a termination handler as a result of the abnormal termination of a try body by a return, or other direct flow of control out of the try body. Abnormal termination occurs whenever control leaves the try body other than by falling through the bottom. This can occur because of a return, goto, continue, or break. It can also occur because of an exception, which presumably cannot be avoided.

In the above example, abnormal termination in the nonexception case can be eliminated easily as follows:

```
/* Correct use of try/finally */
```

```
VOID
function (
    DWORD ... P1,
    .
    .
    DWORD ... Pn
)
{
    try {
        if (...) {
            .
            .
        } else {
            .
            .
        }
    } finally {
        .
        .
    }
    return;
}
```



```
}
```

Now both clauses of the if fall through to the termination handler in all but exceptional cases and execute the termination handler in the most efficient way. This also has the same logical execution as the previous sample.

In summary, the correct use of try/finally is a powerful method to help you write robust applications. Care should be taken to ensure the correct use of try/finally.

INF: Concatenating Resource Files Does Not Work on Windows NT
Article ID: Q83934

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Concatenating resource files (with copy /b or a concatenating program) does not work under Windows NT.

More Information:

Under MS-DOS and OS/2 Presentation Manager (PM), concatenating resources works because each resource file has a header that contains the size of the data in the file. Code that reads the .RES file can read each header to get the size of the data, and then skip to the next resource until the desired resource is located. The .RES file does not have a "master directory" or header of its own.

Under Win32, this method does not work because the CVTRES tool puts a COFF wrapper around the .RES.

INF: RCDATA Begins on 32-Bit Boundary in Windows NT
Article ID: Q84081

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

RCDATA is guaranteed to begin on a DWORD boundary. However, the strings and the integers specified in the statement are not aligned by the Resource Compiler (RC).

More Information:

RCDATA statement:

```
resname RCDATA  
BEGIN
```

```
    0,0,
```

```
END
```

The definition of RCDATA is not changed. The strings and integers specified in the statement, 0 in this case, are not aligned on the DWORD boundary. However, the beginning of the data is DWORD-aligned.

Additional reference words: 32 bit 32-bit double-word double

PRB: Win32s: GetVolumeInformation Returns Incorrect Values
Article ID: Q93639

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

In Win32s 1.0, GetVolumeInformation() always returns a volume ID of 0x12345678.

STATUS

Microsoft has confirmed this to be a problem in Win32s version 1.0. We are researching this problem and will post new information here when it becomes available.

Additional reference words: 3.10

PRB: LIB.EXE: Adding Object Documentation Error
Article ID: Q93641

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Page 233 (Chapter 4, "Library Manager/Adding an Object") of the Microsoft Win32 Preliminary SDK for Windows NT "Tools" manual incorrectly states that adding an object to a library defaults to the given library name as the output library.

However, the following

```
lib32 PROJECT.LIB NEW.OBJ
```

does not output a new PROJECT.LIB file that includes NEW.OBJ; rather, it outputs a file, COFF.OUT, with the new object module, which must be manually renamed to PROJECT.LIB.

The correct syntax is:

```
lib32 -out:PROJECT.LIB PROJECT.LIB NEW.OBJ
```

Additional reference words: 3.10

PRB: Problems Using COMM APIs and the DCB Structure on MIPS
Article ID: Q98887

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

There are problems accessing the DCB (device control block) structure when using the March Win32 Software Development Kit (SDK) for MIPS. The DCB structure is used by the following APIs:

```
BuildCommDCB
BuildCommDCBAndTimeouts
GetCommState
SetCommState
```

The fields declared after `wReserved` appear to be incorrect, as if they should have been shifted down by 2. If any of them are set, then `SetCommState()` returns error 87 (INVALID PARAMETER).

CAUSE

The problems with DCB alignment are due to the fact that the system DLLs were built with the old MIPS compiler (`cc`) and the new MIPS compiler (`mcl`). These compilers have different alignment rules.

RESOLUTION

The workaround is to use the old MIPS compiler (`cc`), which is included on the March beta CD. The new MIPS compiler (`mcl`) is installed by default.

STATUS

Microsoft has confirmed this to be a problem in the March Win32 SDK.

More Information:

The two compilers are described in more detail in the README.TXT file.

The following description of the DCB structure is taken from the help file:

```
typedef struct _DCB { /* dcb */
    DWORD DCBlength; /* sizeof(DCB) */
    DWORD BaudRate; /* current baud rate */
    DWORD fBinary: 1; /* binary mode, no EOF check */
    DWORD fParity: 1; /* enable parity checking */
    DWORD fOutxCtsFlow: 1; /* CTS output flow control */
    DWORD fOutxDsrFlow: 1; /* DSR output flow control */
    DWORD fDtrControl: 2; /* DTR flow control type */
    DWORD fDsrSensitivity: 1; /* DSR sensitivity */
};
```

```

DWORD fTXContinueOnXoff:1; /* XOFF continues Tx */
DWORD fOutX: 1; /* XON/XOFF out flow control */
DWORD fInX: 1; /* XON/XOFF in flow control */
DWORD fErrorChar: 1; /* enable error replacement */
DWORD fNull: 1; /* enable null stripping */
DWORD fRtsControl:2; /* RTS flow control */
DWORD fAbortOnError:1; /* abort reads/writes on error */
DWORD fDummy2:17; /* reserved */
WORD wReserved; /* not currently used */

WORD XonLim; /* transmit XON threshold */
WORD XoffLim; /* transmit XOFF threshold */
BYTE ByteSize; /* number of bits/byte, 4-8 */
BYTE Parity; /* 0-4=no,odd,even,mark,space */
BYTE StopBits; /* 0,1,2 = 1, 1.5, 2 */
char XonChar; /* Tx and Rx XON character */
char XoffChar; /* Tx and Rx XOFF character */
char ErrorChar; /* error replacement character */
char EofChar; /* end of input character */

char EvtChar; /* received event character */
} DCB;

```

Additional reference words: 3.10 3.1

INF: LIB32.EXE Converts Object Files to COFF Format
Article ID: Q93707

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

LIB32.EXE converts .OBJ files into COFF objects before inserting them into a library. Thus, a Microsoft format .OBJ file (non-COFF) inserted into a library will not be the same size if extracted back from the library.

This should not be an issue because the cl386 (x86) or mcl (mips) compilers will produce COFF objects.

Additional reference words: 3.10

INF: Microsoft Implementation of Bit Fields in cl386 Compiler
Article ID: Q88952

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The cl386 compiler deals with bit fields in structures in a manner similar to Microsoft C version 7.0. The following is Microsoft-specific information on bit fields, including 32-bit-specific information, which is clearly marked.

More Information:

Bit Field Types

Microsoft 80x86 compilers allocate bit fields according to the type of the bit field. ANSI requires only the unsigned int/signed int level of functionality, but a Microsoft extension to the ANSI C standard allows char and long types (both signed and unsigned) for bit fields. The compiler will create unnamed bit fields with base type long, short, or char (signed or unsigned) to pad allocated bit fields, forcing alignment to a boundary appropriate to the base type:

char	8 bits
short (int)	16 bits
long	32 bits

Bit Field Packing

The 80x86 compilers merge adjacent bit fields of the same type. This allows you to control the packing with a little more sophistication, as packing is done on the basis of the type you declare the bit field to be. For example

```
struct hello {
    unsigned char a:5;
    unsigned char b:5;
}
```

will align both elements on byte boundaries (as they cannot fit within a byte), whereas

```
struct there {
    unsigned short a:5;
    unsigned short b:5;
}
```

will pack them into 10 bits. Note also that those two structure representations will be padded out to 16 bits to fill the short. Note that

```
struct blankspace {
    unsigned long a:5;
    unsigned long b:5;
}
```

will be allocated in 32 bits due to the long data type of a and b, even though the packed bit fields would fit inside a short. Also,

```
struct mybitfield{
    unsigned short a:5;
    unsigned char b:4
    unsigned long c:6
}
```

will not have its bit fields merged at all, as there are no adjacent bit fields of the same type.

MIPS-Specific Information

Unlike the 80x86 compilers, the MIPS compiler ignores the types of adjacent bit fields with regard to packing. The MIPS compiler packs adjacent bit fields into the smallest integral type, such as char or short, regardless of type.

However, like the 80x86 compilers, nonadjacent bit fields will not be packed.

Bit Field Alignment

Every data object has an "alignment requirement" (AR). For data types that are neither aggregates nor arrays, this is either the size of the object OR the current packing size [that is, /Zp or #pragma pack()], whichever is LESS. The AR of an aggregate or array is the largest AR of its members or elements. Every object is allocated such that:

```
OFFSETOFALLOCATION % AR = 0
```

A bit field in every way has the semantics of its integral type, except that adjacent bit fields are packed into the same allocation unit if the integral types are the same size (the bit fields need not be the same width) and the new bit field will fit without crossing the boundary imposed by the common AR of the bit fields.

80x86 32-Bit-Specific Information

The default data type packing size is 4 for 32-bit targets (/Zp4). Note the example structure mybitfield, above. The structure will be allocated in 8 bytes: 2 for mybitfield.a, 1 for mybitfield.b, 1 byte to pad to a 4-byte boundary, and 4 for mybitfield.c. Likewise, a single character bit field would be allocated in 32 bits (1 byte

for the bit field, and 3 bytes to pad to a 4-byte boundary).

Signed/Unsigned Interpretation

On the 16-bit implementations, unsigned int has the same behavior as unsigned short. This is similar to signed types. Bit fields defined as int are treated as signed.

32-Bit-Specific Information

Note that on the 32-bit implementations, unsigned long has the same behavior as unsigned int, likewise with signed.

Signed bit fields work, but the behavior of signed 1-bit fields is implementation specific (in other words, don't use them). However, longer fields work as expected.

Bit Field Allocation

For 16-bit targets, bit fields default to size short, which can cross a byte boundary but not a 16-bit or 32-bit boundary. If the size and location of a bit field would cause it to overflow the current integer, the field is moved to the beginning of the next available integer. If a bit field is declared as a long, it can hold up to 32 bits. In either case, an individual field cannot cross a 16 or 32-bit boundary.

32-Bit-Specific Information

Bit fields default to size long for the 32-bit compiler. An individual field may cross a 16-bit boundary.

Merged bit fields are allocated within an integer from the least-significant to the most-significant bit. In the following code

```
struct mybit fields {
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test;

void main( void );
{
    test.a = 2;
    test.b = 31;
    test.c = 0;
}
```

the bits would be arranged as follows:

```
0    1    F    2    << Hex value
0000 0001 1111 0010 << Binary value
cccc cccb bbbb aaaa << field allocated corresponding bit
```

Since the 80x86 family of processors store the low byte of 2-byte word values before the high byte, the integer 0x01F2 above would be stored in physical memory as 0xF2 followed by 0x01.

Bit Field Usage

Currently, simple operations with bit fields are encouraged instead of complex operations, as the code produced is relatively more efficient. For instance, bit-field assignment statements are less efficient than corresponding non-bit-field assignments.

In addition, the compiler doesn't currently optimize bit-field initialization constructs; for example

```
    struc.a = 4;  
    struc.b = 6;  
    struc.c = 3;...
```

are not optimized with a single assign. Therefore, bit fields are an efficient way to organize binary data items in a small space, but may not be quite as speed efficient, depending on the operation.

Additional reference words: 3.10

INF: OS/2-to-Windows Migration Information

Article ID: Q89058

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Several tools designed to make it easier to port OS/2 applications to Windows and Windows NT are available in the "Porting from OS/2" download library in the MSWIN32 forum on CompuServe. These tools handle resource-file conversion, help-file conversion, image conversion, and in some cases, source-code conversion. Consider the following as guidelines:

1. Convert the OS/2 .RC file to Windows with the tools in RESCONV.ZIP.
2. Solve build and environment issues. Recompile and link the original code on Windows NT.
3. Run the Sed script on OS/2 code. The Gnu Sed script ported to Windows NT is in SEDGNU.ZIP.
4. Run MS_SSED, which does some custom munging of your source code.
5. Some manual steps are still needed; these are outlined in CHANGE.DOC.

Your application has now been ported to Windows NT.

More Information:

The following is a brief description of the available tools, including two files that were updates to the original posting:

ANNOTE.ZIP	Source code annotation system. It is a batch version of PortTool (Windows NT SDK) with a database for converting OS/2 to Windows. Note that most of the data file for this is outdated with the tools in OS2PORT.ZIP.
APICNT.ZIP	Examines your OS/2 DLL and .EXE, extracts OS/2 API information, and shows you how to analyze this information. This should help you estimate and plan the port.
HELPC.ZIP	Converts OS/2 IPF files to Windows RTF.
IMAGEC.ZIP	Converts OS/2 image files to Windows format.
OS2PORT.ZIP	This tree takes Petzold's HEAD.EXE program from

"Programming the OS/2 Presentation Manager"
(Microsoft Press) and shows how to port it to
Windows NT.

RESCONV.ZIP Convert OS/2 .RC files to Windows .RC files.

ROOT.ZIP This contains a README.TXT and UNPACKME.BAT file,
which gives the complete description of the entire
tree.

SEDGNU.ZIP Gnu Sed. This is NOT copyrighted by Microsoft.

WINHLP.ZIP WINHELP.EXE Help file for moving OS/2 code to
Windows. Contains hypertext links to 16- and 32-bit
Windows Help files.

These files were included in the update to the original posting:

UPDT01.ZIP Provides updates to the files listed above.

SHRMEM.ZIP Provides a DLL that manages shared memory under
Windows NT. This includes provisions to set up the
memory addresses so they are the same across
different processes.

Additional reference words: 3.10

INF: Source-level Debugging Under NTSD

Article ID: Q99053

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.10
-

Summary:

The following are the steps used for source-level debugging under NTSD:

1. Compile using `-Zi` and `-Od`.
2. Link using `debug:full` and `debugtype:coff`.
3. Load the program into the debugger.
4. Use `s+` to change to source mode.

-or-

Use `s&` to change to mixed mode.

For a console application, type `"g main"` to get to the program start.
For a GUI application, type `"g WinMain"` to get to the program start.

Type `"v .<number>"` to list source lines starting at `<number>`. For example, type the following

```
v .20
```

to see all lines starting at line 20.

Additional reference words: 3.10

INF: Preserving Case When Assembling /Fa Listing
Article ID: Q94271

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When assembling the output of a /Fa listing from the Microsoft NT C compiler, it necessary to specify the -Ml switch on the MASM386 command line. The -Ml switch instructs the assembler to preserve case for all symbols. The symbols generated by the C compiler are case sensitive, which makes it necessary for the assembler to preserve case.

If the -Ml switch is not used, errors due to unresolved externals will occur during linking.

Additional reference words: 3.10 3.1

PRB: Debugging the Open Common Dialog Box in WinDbg
Article ID: Q99952

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

SYMPTOMS

When debugging an application that uses the Open common dialog box (created by the GetOpenFileName function), the program stops and the following information is displayed in the Command window:

Thread Terminate: Process=0, Thread=2, Exit Code=1

CAUSE

The Open common dialog box causes a thread to be created. At this point in the debugging, that thread has terminated. By default, WinDbg halts whenever a thread terminates.

RESOLUTION

Execute the go command (type "g" at the command prompt). Execution will continue.

More Information:

To prevent WinDbg from halting when a thread is terminated, select Debug from the Options menu and check "Go on thread terminate."

Additional reference words: 3.10

INF: Win32 Subsystem Object Cleanup

Article ID: Q89290

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Win32 subsystem guarantees that all Win32 objects owned by a process will be freed when an application terminates. To accomplish this, the Win32 subsystem keeps track of who owns these objects; it also keeps a reference count. Reference counts are used when the object is owned by more than one process. For example, a memory mapped file can be used to provide interprocess communication, where more than one process would own that object. The subsystem must make sure that the reference count is zero before the object can be freed.

Freeing of Win32 objects can occur at different times. In general, it occurs at process termination, but for some objects, it occurs at thread termination.

Note: When running Win32 applications with Windows 3.1 using the Win32s environment, it is the responsibility of the Win32 application to ensure that all allocated GDI objects are deleted before the program terminates. This is different from the behavior of the application with Windows NT. With Windows NT, the GDI subsystem cleans up all orphaned GDI objects. Because there is no GDI subsystem with Windows 3.1, this behavior is not supported.

More Information:

At process or thread termination, the Win32 subsystem searches its lists to find objects owned by this process or thread. Those that are owned by the terminating process or thread and whose reference counts will be set to zero when the process or thread is fully terminated will be freed.

The freeing of objects is slightly different for Win32 applications running under Win32s on Windows 3.1. The 16-bit objects (GDI objects, windows, global memory, etc.) follow the same clean-up rules as Win16 applications do under Windows 3.1. The 32-bit objects, such as memory allocated via `VirtualAlloc()`, shared memory via mapped file I/O, 32-bit modules, thunks allocated on the fly (for hook procedures, `wndprocs` etc.) are all handled by Win32s and freed at process termination.

The following is a list of Win32 objects. Note that it may not be complete.

BASE: console, event, file (including file mapping), mutex, semaphore, thread, process, pipe (including named pipes)

GDI: device context (DC), bitmap, pen, brush, font, region, palette

USER: window, cursor, icon, menu, accelerator table, desktop,
DDE communication objects, DDE conversation objects, dialog

INF: Using MFC Build Clean Option

Article ID: Q94272

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

After executing the following command:

```
nmake clean obj=$nwd
```

the following error message is displayed:

```
Unable to find directory wd
```

CAUSE

=====

After rebuilding the Microsoft Foundation Classes (MFC) library, the object files were contained in the default directory \$NWD. The dollar sign character is used in NMAKE to specify the beginning of a macro expansion [such as \$(OBJ)]. This is the reason the dollar sign and the following character were lost when \$NWD was simply specified.

RESOLUTION

=====

There are two methods that can be used to delete the \$NWD directory. The first method is to execute the command:

```
nmake MODEL=N TARGET=W DEBUG=1 clean
```

With this method, the nmake file determines the name of the directory by the build parameters specified.

The second method is as follows:

```
nmake clean OBJ=^$$NWD
```

Additional reference words: 3.10 3.1

INF: Warning C4056: Overflow in Floating Point
Article ID: Q94273

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Microsoft NT C compiler may produce the following warning:

```
test.cpp(12) : warning C4056: overflow in floating point  
constant arithmetic
```

This does not represent a problem in the source being compiled. A future release of the compiler will correct this problem.

Additional reference words: 3.10 3.1

INF: Unicode Conversion to Integers

Article ID: Q89295

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Under Windows NT, strings may be either Unicode or ANSI. There is no function for reliably converting a string that might be either Unicode or ANSI to an integer. This is because, given a random set of bytes, it is difficult to determine whether the string is in Unicode or ANSI. The calling program has to know which format the string uses in order to convert it.

If the string uses Unicode, the functions `wcstol()`, `wcstoul()`, and `wcstod()` can be used to perform the conversion.

Note that when you are using the Win32 application programming interface (API), you can choose what kind of characters you get from the console or window manager. The names of the API functions that are called to use Unicode and ANSI characters are different. For more details, see Chapter 93 in the overview, "Unicode."

To mark a string as Unicode, insert the byte-ordering-mark (BOM) `0xFEFF` in the string and/or file.

More Information:

You can assume that the first 128 bytes in each character set are in the same codepoint. For portability, you should code character conversions in this range as:

```
{
    TCHAR    c;
    ...
    i = c - TEXT('0');
}
```

The `TEXT` macro places an "L" before the constant if Unicode is defined.

INF: How HEAPSIZE/STACKSIZE Commit > Reserve Affects Execution
Article ID: Q89296

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The syntax for the module-definition statements HEAPSIZE and STACKSIZE is as follows

```
HEAPSIZE [reserve] [,commit]
STACKSIZE [reserve] [,commit]
```

The remarks for HEAPSIZE and STACKSIZE on page 236 of the "Tools" manual that comes with the Win32 Preliminary SDK for Windows NT state the following:

When commit is less than reserve, memory demands are reduced but execution time is slower.

By default, commit is less than reserve.

The reason that execution time is slower (and it is actually only fractionally slower), is that the system sets up guard pages and could have to process guard page faults.

More Information:

If the committed memory is less than the reserved memory, the system sets up guard page(s) around the heap or stack. When the heap or stack grows big enough, the guard pages start accessing outside the committed area. This causes a guard page fault, which tells the system to map in another page. The application continues to run as if you had originally had the new page committed.

If the committed memory is greater than the reserve, no guard pages are created and the program faults if it goes outside the committed memory area.

Experimenting with the commit versus reserve numbers may result in a combination that would produce noticeable results, but for most applications, this difference is probably not noticeable. The potential benefits do not warrant significant experimentation.

Additional reference words: 3.10

INF: MFC TRACE Output Not Working

Article ID: Q94274

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

For TRACE output to work correctly, a file in the \MFC\SRC directory named AFX.INI needs to be copied to the \WINNT directory. This file is necessary for TRACE to work properly.

Note: If TraceEnable = 1, then running any MFC application with Debug information will be slow.

Additional reference words: 3.10 3.1

INF: WinDbg
Article ID: Q99953

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The WinDbg message "Breakpoint Not Instantiated" indicates that the debugger could not resolve an address. This can happen for several reasons:

- A specified symbol does not exist. In this case, check for misspelling and check the state of the "ignore case" option if the symbol contains mixed case.
- or-
- The symbol exists, but the EXE or DLL was built with the wrong debugging information, or none at all. Use the -Zi and -Od compiler options and use the -debug:full and -debugtype:cv linker options.
- or-
- The symbol exists, but it is in a module that has not yet been loaded. If the symbol is in a DLL that is dynamically loaded the breakpoint was probably set before the DLL was loaded. The message is harmless, because WinDbg will instantiate the BP when the module is loaded.

Additional reference words: 3.10

INF: Fatal Error C1001: ICE ('msc1.cpp', line 555)
Article ID: Q94319

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following error is generated if a function that returns a C++ class is declared extern "C":

```
fatal error C1001: INTERNAL COMPILER ERROR (compiler file 'msc1.cpp',  
line 555)
```

Microsoft compilers do not currently support functions declared extern "C" and returning C++ classes.

More Information:

The following code fragment demonstrates the internal compiler error:

Sample Code

```
-----  
  
class C1;  
  
extern "C"  
{  
    C1 func1();  
}  
  
class C1  
{  
    public:  
    int ok() { return 0; }  
};
```

Additional reference words: 3.10 3.1

INF: Windows NT Compiler Always Includes chkstk()
Article ID: Q94320

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Windows NT C compiler seems to add `chkstk()` calls even when the `/Gs` switch is used to disable stack overflow checking.

This behavior is by design. `chkstk()` is used to allocate frame sizes greater than or equal to 4K. This ensures that the application does not access beyond the stack guard page.

Additional reference words: 3.10 3.1

PRB: Windows NT: Inline Assembly Code Generation Error
Article ID: Q95163

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

The following piece of code

```
class UINT64
{
    public:
        ...

        UINT32  m_low;
        UINT32  m_high;
};

UINT64 n1;
double n2;

_asm
{
    fild qword ptr n.m_low
    fild n2
}
```

compiles correctly on x86 platforms, but the disassembly is as follows:

```
FILD DWORD PTR [n]
FILD QWORD PTR [n2]
```

CAUSE

This is a known code generation problem with the C/C++ compiler shipped with the Windows NT Software Development Kit (SDK) for x86 platforms.

RESOLUTION

Microsoft is researching this problem and will post new information here in the Microsoft Knowledge Base as it becomes available.

More Information:

Note that if you choose to include inline assembly into your code, you lose ease of portability between platforms.

Additional reference words: 3.10 3.1

INF: Fatal Error C1056: Out of Macro Expansion Space
Article ID: Q94321

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The maximum expanded size of a macro under the NT compiler is 16K. If a macro exceeds this limit, the following error message will be generated:

```
fatal error C1056: compiler limit : out of macro expansion space
```

There is no workaround to this limit on the size of the macro expansion buffer.

Additional reference words: 3.10 3.1

INF: Fatal Error C1001: ICE (file 'mscl.cpp', line 555)
Article ID: Q94322

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When the `_export` keyword is used to mark a C++ class for export, the following errors are generated:

```
test.cpp(7) : error C2165: '_far' : cannot modify pointers to data
test.cpp(7) : fatal error C1001: INTERNAL COMPILER ERROR
             (compiler file 'mscl.cpp', line 555)
             Contact Microsoft Product Support Services
```

To correct this problem, add a `_near` keyword after the `_export` keyword.

More Information:

The following code fragment generates the internal compiler error:

```
class _export ExportedClass
{
public:
    ExportedClass();
    virtual void Member();
    static void SMember();
};
```

Additional reference words: 3.10 3.1

INF: Setting Dynamic Breakpoints in WinDbg
Article ID: Q100642

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The WinDbg breakpoint command contains a metacircular interpreter; that is, you can execute commands dynamically once a breakpoint is hit. This allows you to perform complex operations, including breaking when an automatic variable has changed, as described below.

The command interpreter of WinDbg allows any valid C expression to serve as a break condition. For example, to break whenever a static variable has changed, use the following expression in the Expression field of the breakpoint dialog box:

```
&<variablename>
```

In addition, the length should be specified as 4 (the size of a DWORD) in the length field.

This technique does not work for automatic variables because the address of an automatic variable may change depending on the value that the stack pointer has upon entering the function that defines the automatic variable. This is one case where the breakpoint needs to be redefined dynamically.

For this purpose, a breakpoint can be enabled at function start and disabled at function exit, so that the address of the variable is recomputed.

More Information:

Suppose that the name of the function is "subroutine" and the local variable name is "i". The following steps will be used:

1. Start the program and step into the function that defines the automatic variable with the commands:

```
g subroutine
p
bp500 ={subroutine}&i /r4 /C"?i"
```

The breakpoint number is chosen to be large so that the breakpoint will be well out of range of other breakpoints. Note that /r4 indicates a length of 4 because i is an integer. Make this number larger for other data types. The command "?i" prints out the value of i.

2. Next, disable this first breakpoint with the command

```
bd500
```

because the address of `i` may change. The breakpoint will be enabled when in the scope of function subroutine.

3. The second breakpoint definition is set at the entry point of the function:

```
bp .<FirstLine> /C"be 500;g"
```

This is where the breakpoint is enabled. Note that `<FirstLine>` is the line number of the first statement in the function subroutine.

4. The last breakpoint is set at the end of the function

```
bp .<LastLine> /C"bd 500;g"
```

and will disable the breakpoint again. Note that `<LastLine>` is the line number of the last statement in the function subroutine.

Note that if the function has more than one exit point, multiple breakpoints may have to be defined.

Program execution stops when breakpoint #500 is hit (for example, the value of `i` changes), but execution will continue after the other two breakpoints because they contain go ("`g`") commands.

Additional reference words: 3.10

INF: Base Date for Time Differs Between NT and C/C++ 7.0
Article ID: Q89580

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In versions 6.0 and earlier of Microsoft C, the time functions returned the time since 00:00:00, 01-01-1970 universal coordinated time (UCT).

During the development of Microsoft C/C++ version 7.0, the basis for the time functions was mistakenly changed to 00:00:00, 12-21-1899 UCT.

Microsoft did not become aware of the problem in time to correct it before the final product release. Therefore, the base date for the time functions in Microsoft C/C++ 7.0 is 00:00:00, 12-31-1899 UCT.

However, corrections were made in the Windows NT SDK. Its time functions once again use 00:00:00, 01-01-1970 UCT as a basis. Therefore, if a date is returned by a C/C++ 7.0 time function and interpreted by an NT SDK time function (or vice versa), the time will be roughly 70 years off. This is also true if the Microsoft Foundation Classes (MFC) are used (specifically, CTime).

The correction will also be made in a subsequent release of the C/C++ compiler.

More Information:

To compensate for this difference:

1. Subtract 2,209,075,200 from the value returned by a C/C++ time function so that it will be compatible with an NT SDK function.
2. Add 2,209,075,200 to the value returned by an NT SDK function so that it will be compatible with a C/C++ time function.

The calculations to determine this figure are

$$2,209,075,200 \text{ seconds} = 86,400 \text{ seconds/day} * 25,568 \text{ days}$$

$$25,568 = (70 \text{ years} * 365 \text{ days/year}) + 17 \text{ days} + 1 \text{ day}$$

where "17" is the number of leap days and "1" is the day added to compensate for the difference between 12-31 and 01-01.

Note that UCT (universal coordinated time) is synonymous with Greenwich mean time (GMT).

INF: CTYPE Macros Function Incorrectly

Article ID: Q94323

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When an application that is linked to CRTDLL.LIB is compiled without defining `_MT` and `_DLL`, the CTYPE.H family of macros will not operate correctly.

To define `_MT` and `_DLL` on the CL386 command line, just add the following to the command line:

```
-D_MT -D_DLL
```

By adding these defines, the CTYPE macros will be properly initialized.

Additional reference words: 3.10 3.1

INF: Calling Conventions Supported by the 32-Bit Compiler
Article ID: Q89691

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

There are three calling conventions supported by the Windows SDK 32-bit compiler. They are C (`_cdecl`), Standard (`_stdcall`), and Fastcall (`_fastcall`). The Pascal calling convention (`_pascal`), which was supported by Microsoft's 16-bit compilers, is not supported.

The following table summarizes the calling conventions:

	<code>_cdecl</code>	<code>_stdcall</code>	<code>_fastcall</code>
Arguments	Pushed R to L	Pushed R to L	Note 3
Stack cleaned up by	Caller	Callee	Callee
Naming convention	Prepend <code>_</code>	Note 1	Note 2

Note 1: `_stdcall` decorates the name by prepending an underscore (as in `_cdecl`) and then appending an at sign (@) and the decimal representation of the number of bytes to be pushed on the stack. All arguments are widened to a multiple of 4 bytes.

Note 2: `_fastcall` decorates the name by prepending an at sign (@) and appending an at sign (@) and the decimal representation of the number of bytes to be pushed on the stack. All arguments are widened to a multiple of 4 bytes.

Note 3: The first two arguments of size `DWORD` or smaller are placed in registers. The remainder of the parameters are pushed on the stack from right to left. This is not guaranteed to be true in future versions.

The default calling convention is `_cdecl`. If a function declared as `_fastcall` or `_stdcall` takes a variable number of arguments, the `_cdecl` calling convention is used.

Part of the `_stdcall` convention is that `EAX`, `ECX`, and `EDX` are not preserved across function calls while `EBX`, `ESI`, `EDI`, and `EBP` are preserved by the calling function.

MORE INFORMATION

=====

The following samples illustrate the C calling convention from the perspectives of the caller and the callee:

```
int _cdecl CFunc( int a, int b );
```

```

    (caller)                (callee)

push   b                    _CFunc PROC NEAR
push   a                    .
call   _CFunc               .
add    esp,8                .
.      .                    RET
.      .                    _CFunc ENDP
.

```

```
int _cdecl CVarFunc( int a, ... );
```

```

    (caller)                (callee)

push   ...                  _CVarFunc PROC NEAR
push   a                    .
call   _CVarFunc           .
add    esp,4+...           .
.      .                    RET
.      .                    _CVarFunc ENDP
.

```

The following sample illustrates the Standard calling convention from the perspectives of the caller and the callee:

```
int _stdcall StdFunc( int a, int b );
```

```

    (caller)                (callee)

push   b                    _StdFunc@8 PROC NEAR
push   a                    .
call   _StdFunc@8          .
.      .                    .
.      .                    RET 8
.      .                    _StdFunc@8 ENDP

```

The following sample illustrates the Fastcall calling convention from the perspectives of the caller and the callee:

```
int _fastcall FastFunc( int a, int b );
```

```

    (caller)                (callee)

mov    edx, b               @FastFunc@8 PROC NEAR
mov    ecx, a               .
call   _FastFunc           .
.      .                    .
.      .                    RET 8
.      .                    @FastFunc@8 ENDP

```

Additional reference words: 3.10

PRB: Debugging an Application Driven by MS-TEST

Article ID: Q100957

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

SYMPTOMS

When an application driven by MS-TEST is being debugged by WinDbg or NTSD (for example, after an exception has occurred), both the application and the debugger hang.

CAUSE

The debugger is hooked and ends up hanging.

RESOLUTION

It is not possible to use NTSD or WinDbg to debug an application that is driven by MS-TEST. Use Dr. Watson (drwtsn32) instead. Note that you must turn off Visual Notification.

STATUS

Microsoft has confirmed this to be a problem in Windows NT 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

INF: Format for LANGUAGE Statement in .RES Files
Article ID: Q89822

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The syntax for the LANGUAGE statement in the resource script file is given as follows on page 378 of the Win32 Preliminary SDK "Tools" manual:

```
LANGUAGE major, minor
```

```
major
```

```
Language identifier. Must be one of the constants from WINNLS.H
```

```
minor
```

```
Sublanguage identifier. Must be one of the constants from WINNLS.H
```

For example, suppose that you want to set the language for the resources in a file to French. For the major parameter, you would choose the following constant from the list of language identifiers

```
#define LANG_FRENCH                0x0c
```

and you would have the choice of any of the sublanguages that begin with SUBLANG_FRENCH in the list of sublanguage identifiers. They are:

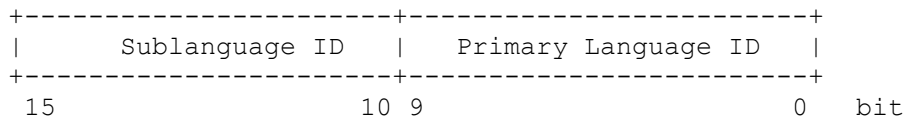
```
#define SUBLANG_FRENCH              0x01  
#define SUBLANG_FRENCH_BELGIAN     0x02  
#define SUBLANG_FRENCH_CANADIAN    0x03  
#define SUBLANG_FRENCH_SWISS       0x04
```

RC.EXE does not directly place these constants in the .RES file. It uses the macro MAKELANGID to turn the parameters into a WORD that corresponds to a language ID.

More Information:

The following information is taken from the WINNLS.H file.

A language ID is a 16-bit value that is the combination of a primary language ID and a secondary language ID. The bits are allocated as follows:



Language ID creation/extraction macros:

MAKELANGID - Construct language ID from primary language ID and
sublanguage ID.
PRIMARYLANGID - Extract primary language ID from a language ID.
SUBLANGID - Extract sublanguage ID from a language ID.

The macros are defined as follows

```
#define MAKELANGID(p, s)      (((USHORT)(s) << 10) | (USHORT)(p))  
#define PRIMARYLANGID(lgid) ((USHORT)(lgid) & 0x3ff)  
#define SUBLANGID(lgid)     ((USHORT)(lgid) >> 10)
```

Additional reference words: 3.10

INF: Microsoft NT C++ Is AT&T 2.1 Compatible
Article ID: Q94324

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT
version 3.1
-

Summary:

The Microsoft NT C/C++ Compiler is AT&T 2.1 compatible.

Additional reference words: 3.10 3.1

INF: Usage of the afx_msg Type

Article ID: Q94325

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Microsoft Foundation Classes (MFC) samples include the "afx_msg" type in the declaration of all message processing functions. This string equates to null in both the Windows 3.1 and Win32 AFXWIN.H header files.

The afx_msg type has no syntactic value; rather, it is a visual cue that the function is used as a message handler. It is good practice to use this type in derived classes.

Additional reference words: 3.10 Win3.1

INF: Tips for Writing Multiple-Language Scripts

Article ID: Q89865

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

To aid you in writing multiple-language resources, the Win32 development system supports language scripts. To create a multiple language script, first create a single-language script file (American English, for example), and duplicate the translations in your script file. You need a complete translation only once for each major language. Only those resources that have differences between the major language and the sublanguage need be included in the sublanguage areas of the script. The system will use the main language resource if it doesn't find a resource for the sublanguage.

Sample Code

```
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_US
<original script file>
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_UK
<portions of script file that are different for UK>
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_AUS
<portions of script file that are different for Australia>
LANGUAGE LANG_ENGLISH,SUBLANG_ENGLISH_CAN
<portions of script file that are different for Canada>
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH
<entire script file translated to French>
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH_CAN
<portions of script file that are different for Canada>
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH_SWISS
<portions of script file that are different for Switzerland>
LANGUAGE LANG_FRENCH,SUBLANG_FRENCH_BELGIAN
<portions of script file that are different for Belgium>
LANGUAGE LANG_SPANISH,SUBLANG_SPANISH
<entire script file translated to Spanish>
```

Additional reference words: 3.10 windows NT

INF: Writing Multiple-Language Resources

Article ID: Q89866

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When you are writing multiple-language resources, the dialog box identifiers need to be identical for each language instance, as demonstrated below.

```
#define DialogID          100

DialogID DIALOG  0, 0, 210, 10
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
.
.
.
DialogID DIALOG  0, 0, 210, 10
LANGUAGE LANG_FRENCH, SUBLANG_FRENCH
```

The FindResource() application programming interface (API) function is used by the system to fetch the dialog box. FindResource() gets the locale information for the process, then attempts to fetch the resource with that language identifier using FindResourceEx(), the language-specific API function for fetching resources. If FindResourceEx() fails to load the language-specific dialog box, FindResource() then attempts to load the neutral dialog box, which should fetch LANG_FRENCH, SUBLANG_FRENCH, if the locale is SUBLANG_FRENCH_CAN or similar.

The LANGUAGE identifiers and the VERSIONINFO language identifiers should also be identical. The code page for resources is always the Unicode code page. The system will translate from Unicode to the required code page.

The preferred method of developing multiple-language resources is to include a LANGUAGE statement for each language supported rather than using the CODEPAGE, LANGUAGE identifier, and VERSIONINFO information. Although the CODEPAGE information will work, the new method is easier to use.

INF: Using Communal Variables in MASM386
Article ID: Q94327

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

It is possible to create two object modules that share a communal variable.

More Information:

The MASM386 COMM statement can be used to establish communal variables between modules. COMM declares a variable external and instructs the linker to define the variable if it has not been explicitly defined in a module.

Below is an example that demonstrates how a variable can be initialized in one module and at the same time be communal with another module.

MODULE1.ASM
=====

```
    PUBLIC    X
    _DATA    SEGMENT DWORD USE32 PUBLIC 'DATA'
    X        DW      1234H
```

MODULE2.ASM
=====

```
    _DATA    SEGMENT DWORD USE32 PUBLIC 'DATA'
    COMM     X:WORD
```

MODULE1 merely defines X in the normal fashion and initializes it to 1234H. MODULE2 declares X communal using the COMM statement. Please note that the COMM statement is not required in MODULE1.

Additional reference words: 3.10 3.1

INF: Default Alignment of Structures and Classes

Article ID: Q94328

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Microsoft NT C Compiler aligns structures and classes on 32-bit boundaries by default; however, this can be changed with the /Zp compiler switch. The /Zp switch allows you to align structures and classes to a specific byte boundary.

For example, the following aligns structures and classes on 1-byte boundaries:

```
CL386 -Zp1 test.c
```

For more information on -Zp or other compiler switches, see the Microsoft Win32 "Tools" manual.

Additional reference words: 3.10 3.1

INF: Enabling Disk Performance Counters

Article ID: Q100289

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

By default, disk performance counters are not started. Therefore, almost all of the disk metrics obtained will be zero. To enable the disk performance counters, execute the following command and then reboot the system:

```
diskperf -y
```

DISKPERF.EXE is located in %SystemRoot%\system32.

The decision to avoid starting disk performance counters by default was made for performance reasons.

Additional reference words: 3.10

PRB: MS-SETUP Uses \SYSTEM Rather Than \SYSTEM32
Article ID: Q98888

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

Call GetWindowsSysDir() in the SETUP.MST file. The return value is C:\WINNT\SYSTEM\ instead of C:\WINNT\SYSTEM32\.

CAUSE

Windows on Windows (WOW) returns the SYSTEM directory, not the SYSTEM32 directory, to 16-bit applications such as MS-SETUP. This is done for compatibility reasons.

RESOLUTION

Determine whether the setup code is being run under WOW or Windows version 3.1 by checking the WF_WINNT bit (0x4000) in the return from GetWinFlags(). Choose either the return from GetWindowsSysDir() or <winows dir>\system32 as appropriate.

More Information:

Note that there are additional considerations for network installs for Win32s, because the SYSTEM directory may not be a branch off of the Windows directory.

Additional reference words: 3.10 3.1

PRB: Selecting Overlapping Controls in Dialog Editor

Article ID: Q90384

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

Create a dialog box using the Dialog Editor. Place a button onto the dialog box. Create a frame and place it so that it encompasses the button. It is not possible to select that button with the mouse. However, if the frame is created before the button and then moved or placed over the button, then it is possible to select either the frame or the button.

CAUSE

This behavior is by design. When controls are overlapped, the control that is selected when the mouse is clicked is the one that comes last in Z-order.

As a special case, it is possible to select a control placed "underneath" a group box.

RESOLUTION

From the Arrange menu, choose Order/Group. This will bring up a dialog box. Change the Z-order of the button to be after that of the frame. The Z-order may also be changed by manually editing the resource file. The controls that are further down in the file will be "on top."

Note: If the frame is selected and is on top of the button, pressing SHIFT+TAB selects the previous control, which will be the button. This does not allow the position of the control to be changed with the mouse; however, it does allow the text and size to be changed.

Additional reference words:

PRB: Data Section Names Limited to Eight Characters

Article ID: Q100292

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

SYMPTOMS

Data sections can be named by using #pragma data_seg. This method is commonly used so that the named data sections can be shared using the SECTIONS statement in the DEF file. However, if the length of the name specified in the pragma exceeds eight characters, then the section is not properly shared.

CAUSE

The cl386 compiler truncates the section name length to eight characters because the Win32 Software Development Kit (SDK) linker does not support sections with longer names. Therefore, the names specified in the DEF file do not resolve to actual section names.

STATUS

The longer names require use of the COFF strings table, so the rewrite is not trivial. When the linker in a future Microsoft C/C++ product supports this, then the compiler will as well.

More Information:

Note that in addition, the first character of a section name must be a period. Therefore, the section name, as specified in both the pragma and the DEF file, can be a maximum of a period followed by seven characters.

For more information on the shared named-data section, query on the following words in the Microsoft Knowledge Base:

specify and shared and nonshared and data and dll

Additional reference words: 3.10

INF: Retrieving the CMDIChildWnd Parent Window

Article ID: Q101184

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

This information applies to the Microsoft Foundation Class (MFC) Library.

When creating a `CMDIChildWnd`, the parent of the multiple document interface (MDI) window is a "MDICLIENT" control. The handle to the parent frame window can be retrieved in the following way:

```
hwnd = ::GetParent(lpcreatestruct.hwndParent);
```

Alternatively, use the following

```
CMDIFrameWnd* pFrame = (CMDIFrameWnd*)GetParentFrame();
```

to acquire a pointer to the parent MDI frame window object.

NOTE: `GetParent` is declared to return a pointer to a `CFrameWnd` object, and therefore it is necessary to typecast the result to a `CMDIFrameWnd` object.

Additional reference words: 3.10

INF: MIPS Compiler Does Not Support `__inline`
Article ID: Q90503

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MIPS compiler does not support the `__inline` C++ construct. The compiler doesn't specifically indicate this to the user; instead, the user typically notices numerous syntax errors. The keyword is used in function prototypes, so the syntax errors are often regarding the first curly brace (`{}`) in the first function declaration following the prototype. Search the source code for `__inline` to determine whether this is the reason for the syntax errors.

To test whether `__inline` is the problem without searching the source, see if the code compiles cleanly on an x86 system. If it does, yet several or dozens of syntax error messages result when compiling on MIPS, using `__inline` could be the problem.

Additional reference words: 3.10 3.1

INF: Memory Management Via Malloc()

Article ID: Q90531

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In the Microsoft C run-time for NT, malloc() and its related functions map directly to the system heap routines. They don't do the sub-allocation that they do under MS-DOS/Windows. In many ways, using the C run-time under NT is the same as using the Win32 API because it calls the Win32 API directly.

The C run-time heap manager searches for the first available block of memory that is large enough to satisfy the request. When more memory is needed, the heap manager will grow the heap. When a block of memory is freed, the manager checks the previous and following block to see whether either or both can be combined with the newly freed block to form a contiguous free block. In either case, freed memory is re-used if possible.

More Information:

Information such as block size and status (used or free) is kept in a header in the first few bytes at the beginning of the block, not in a separate location, such as a table.

Problems occur if you corrupt the header information of a block by writing beyond the previous allocated block. When the block with the corrupted header is freed, the heap manager will consider the memory that the header happens to point to as the next block when it attempts to combine adjacent free blocks. The system will crash with an exception in RtlExAllocateHeap upon the next malloc or free involving this memory.

The blocks are contiguous, so overwriting one block when writing to a different block is not considered writing outside the program's memory space. The C run-time heap manager doesn't make sure that writes do not extend beyond the end of an allocated block. This would be very time-consuming considering how often the heap is used.

Note that the same problem can occur with APIs such as LocalAlloc() and GlobalAlloc().

Additional reference words: 3.10 3.1

INF: Using Cout in an Application and DLL
Article ID: Q101185

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The standard output stream (cout) can be used in an application or dynamic-link library (DLL) to display information. When using cout in an application and a DLL simultaneously, special care is needed to ensure that the text display is synchronized.

Because the cout text stream is buffered, text from a DLL may not appear correct relative to text from the application. To correct this problem, always use the endl manipulator when outputting text with cout. Below is an example:

```
cout << "Hello World" << endl;
```

The endl manipulator inserts a newline character and then flushes the stream buffer. This ensures that text displayed from the application and DLL is synchronized.

Additional reference words: 3.10 iostream

INF: Interpreting Executable Base Addresses

Article ID: Q101187

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

LINK32.EXE can be used to dump the portable executable (PE) header of an executable file. Below is a fragment of a dump:

```
7300 address of entry point
7000 base of code
B000 base of data
----- new -----
10000 image base
```

The "image base" value of 10000 is the address where the program begins in memory. The value associated with "base of code," "base of data," and "address of entry point" are all offsets from the image base.

Additional reference words: 3.10

INF: Calculating String Length in Registry

Article ID: Q94920

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When writing a string to the registry, you must specify the length of the string, including the terminating null character (`\0`). A common error is to use `strlen()` to determine the length of the string, but to forget that `strlen()` returns only the number of characters in the string, not including the null terminator.

Therefore, the length of the string should be calculated as:

```
strlen( string ) + 1
```

Note that a `REG_MULTI_SZ` string, which contains multiple null-terminated strings, ends with two (2) null characters, which must be factored into the length of the string. For example, a `REG_MULTI_SZ` string might resemble the following in memory:

```
string1\0string2\0string3\0laststring\0\0
```

When calculating the length of a `REG_MULTI_SZ` string, add the length of each of the component strings, as above, and add one for the final terminating null.

Additional reference words: 3.10 3.1

INF: Order of Object Initialization Across Translation Units
Article ID: Q101188

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The order of initialization of global objects is undefined across translation units. For example, if your application consists of three C++ modules and each module declares an object of xyz class, there is no guarantee during program initialization which of these objects will be constructed first.

In addition to not relying on the order of initialization, you should not use one object's address in another object's initialization when the two objects are contained in different translation units.

AT&T 2.1 does not define the order of initialization for global objects across translation units. The order of initialization is implementation-dependent.

Additional reference words: 3.10

INF: Changes to sprintf/wsprintf Formatting
Article ID: Q90834

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

To stay in compliance with ANSI standards and to remain consistent with the C run-time sprintf routines, some changes have been made to the sprintf and vsprintf routines in the way they interpret various formatting characters. These changes affect only character and string formatting specifiers.

The following table shows the format specifiers and how they're interpreted by the sprintf/vsprintf routines:

Specifier	sprintfA	sprintfW
%c / %s	CHAR / LPSTR	WCHAR / LPWSTR
%C / %S	WCHAR / LPWSTR	CHAR / LPSTR
%hc / %hs	CHAR / LPSTR	CHAR / LPSTR
%lc / %ls	WCHAR / LPWSTR	WCHAR / LPWSTR
%wc / %ws	WCHAR / LPWSTR	WCHAR / LPWSTR

The explicit size specifiers ("h", "l", and "w") take precedence over the case of the actual format specifier ("c" or "s"). Thus, sprintfA(ansibuf, "%hC", param1) means that param1 is interpreted as an ANSI character.

Two items to pay particular attention to:

1. %c and %s act as the generic specifiers and thus change meaning between sprintfA and sprintfW. Most code today that calls sprintfW assumes that %c and %s always indicate ANSI. If you want ANSI, then use %hc and %hs.
2. The earlier %tc and %ts specifiers are no longer supported.

Additional reference words: 3.10 3.0

INF: %S, %B, %C, printf() Format Specifier Changes
Article ID: Q94921

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

In the next release of the Win32 Software Development Kit (SDK), in order to be compatible with ANSI C, the %S printf() format specifier used to print counted STRINGS has been changed to %Z. In the new C run time, %S will be equivalent to %ws.

Also, support for the %B and %C format specifiers has been removed (these format specifiers were once used to print bugcodes and status codes symbolically, but have behaved similar to %X for the past two years).

The following table summarizes the changes:

Old	New
String	String
-----	-----
%wS	%wZ
%S	%Z
%lB	%X
%B	%X
%lC	%X
%C	%X

Additional reference words: percent 3.10 3.1

INF: Getting Windows NT Executable Header Information

Article ID: Q91132

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The executable header information for MS-DOS, OS/2, and Windows executables and DLLs has been provided by EXEHDR.EXE. The following command will get the header information for a Windows NT executable (.EXE or .DLL file):

```
link32 -dump -headers filename.{exe|dll}
```

Information about the exports and imports can be obtained through the following commands:

```
link32 -dump -exports
link32 -dump -imports
```

Additional reference words: 3.10 3.1

INF: WinMain() Arguments in Unicode
Article ID: Q90912

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The prototype for WinMain() is as follows:

```
int PASCAL WinMain(  
    HANDLE hInstance,  
    HANDLE hPrevInstance,  
    LPSTR lpCmdLine,  
    int nCmdShow );
```

The third parameter is an LPSTR, which specifies an ANSI string. WinMain() cannot be defined to accept Unicode input because there is no way for the system to know whether or not the application wants Unicode at the time WinMain() is called; the system knows once the application has registered a window class.

To get the arguments in Unicode, use GetCommandLine().

Additional reference words: 3.10 3.1

INF: Using volatile to Prevent Optimization of try/finally
Article ID: Q91149

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The following is an example of a valid optimization that may take programmers by surprise.

1. A variable (temp) used only within the try-except body is declared outside it, and therefore is global with respect to the try.
2. Assignment to the variable (temp) is in the program only for a possible side effect of doing a read memory access through the pointer.

For example:

```
VOID
puRoutine( PULONG pu )
{
    ...
    ULONG temp;        // Just for probing
    ...
    try {
        temp = *pu;    // See if pu is a valid argument
    }

    except {
        // Handle exception
    }
}
```

The compiler optimizes and eliminates the entire try-except statement because temp is not used later.

If the value of temp were used globally, the compiler should treat the assignment to temp as volatile and do the assignment immediately even if it is overwritten later in the body of the try. The reasoning is that, at almost any point in the try body, control may jump to the except (or an exception filter). Presumably the programmer accessing the variable outside the try wants to get the current (most recently assigned) value.

The way to prevent the compiler from performing the optimization is:

```
temp = (volatile ULONG) *pu;
```

If a temporary variable is not needed, given the example, the read access should still be specified as volatile, for example:

*(volatile PULONG) pu;

Additional reference words: 3.10

INF: Postmortem Debugging Under Windows NT
Article ID: Q94924

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Win32 Software Development Kit (SDK) does not have a postmortem utility designed to ship with the product. However, the system provides fundamental support to enable postmortem analysis of processes and allows tool vendors to create their own postmortem solutions.

A key feature provided in the system is the ability to start up a debugger when an unhandled exception occurs. The debugger can attach to the process after the exception has occurred.

In the first commercial release of Windows NT, a 32-bit version of Dr. Watson is automatically run as a post-mortem debugger when unhandled exceptions occur. The Dr. Watson log obtained from application end-users can be useful to the developers of the application in determining the cause of failure.

When the Win32 SDK is installed, the default post-mortem debugger is changed to be WinDbg, so on the machines developers use to develop/test their applications under development, the developer is immediately given a WinDbg session to investigate the cause of failure when an unhandled exception occurs.

The option also allows administrators to configure systems such that any third-party utility or utility written in-house could be started. The utility is exec'd with debugger/buggee relationship. The process address space is retained and the debugger is set at the instruction causing the unhandled exception.

Thus, this interface is not limited to debuggers, but to any postmortem utility. Therefore, there is an opportunity for tool vendors to provide a myriad of postmortem analysis utilities, some of which could be tied to compiler vendor products (for symbolic dumps and so forth).

A postmortem utility could:

1. Dump memory image to disk for later analysis.
2. View/dump memory structures, memory maps, and so forth via VirtualQuery().
3. Study/dump the CPU state, coprocessor state, and so forth.
4. Take advantage of any application symbolic debugging information if present.

There are many other aspects to a postmortem utility that could be added. Core dumps have additional features (such as system support for reloading)

that will not be provided in the Windows NT final release for this version,
but will be provided in a future release.

Additional reference words: 3.10 3.1 post-mortem 3rd

INF: Use of DLGINCLUDE in Resource Files

Article ID: Q91697

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Windows 3.1 SDK dialog editor needs a way to know what include file is associated with a resource file that it opens. Rather than prompt the user for the name of the include file, the name of the include file is embedded in the resource file in most cases.

Embedding the name of the include file is done with a resource of type RCDATA with the special name DLGINCLUDE. This resource is placed into the .RES file and contains the name of the include file. The dialog editor looks for this resource when it loads a .RES file. If this resource is found, then the include file is opened also; if not, the editor prompts the user for the name of the include file.

In some Windows 3.1 build environments, the dialog editor was used to create dialogs that were placed in more than one .DLG file. These different .DLG files were then included in one .RC file, which was compiled with the resource compiler. Therefore, the resource file gets multiple copies of a RCDATA type resource with the same name, DLGINCLUDE, but the resource compiler and dialog editor don't complain.

In the Win32 SDK, changes were made so that this resource has its own resource type; it was changed from an RCDATA-type resource with the special name, DLGINCLUDE, to a DLGINCLUDE resource type whose name can be specified. The dialog editor would look for resources of the type DLGINCLUDE.

Changes were made to CvtRes so that it gives an error if it finds a resource that has the same type, name, and language as another resource in the file. We are being more strict about the need for resources to be unique in the Win32 SDK than the Windows 3.1 SDK. This is good because there was never any guarantee at run time as to which of the two or more resources would be returned by LoadResource().

This means that some applications being ported to Windows NT give an error when their resources are compiled because they have duplicate RCDATA type resources with the same name (DLGINCLUDE). This error is by design. The workaround is straightforward: delete all the DLGINCLUDE RCDATA type resource statements from all the .DLG files.

Finally, because it does not make sense to have the DLGINCLUDE type resources in the executable, CvtRes will strip them out so that they don't get linked into the EXE.

Additional reference words: 3.10 3.1

INF: Warning 0505: No Modules Extracted from 'FILENAME.LIB'
Article ID: Q92503

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The linkers that shipped with the preliminary releases of the Win32 SDK for Windows NT issue the following warning if they are given a library that is not used during linking

```
warning 0505: no modules extracted from 'filename'.lib
```

There is no way to disable the generation of this warning. To avoid the warning, do not give the linker the names of libraries that cause it to be generated. The ability to disable the warning is being considered for a later version of the linker.

More Information:

The SDK sample makefiles all use constants defined in \H\NTWIN32.MAK to specify the libraries that should be included in the link line for a particular kind of application. The constants are:

conlibs	guilibs	psxlibs
conlibsmt	guilibsmt	
conlibsdll	guilibsdll	

The MFC samples use MFC\SAMPLES\NTSAMPLE.MAK. The definition of CONLIBS and GUILIBS differs between NTWIN32.MAK and NTSAMPLE.MAK:

```
conlibs: libc.lib ntdll.lib kernel32.lib netapi.32
guilibs: libc.lib ntdll.lib kernel32.lib user32.lib gdi32.lib \
  winspool.lib comdlg32.lib advapi32.lib olecli32.lib olesvr32.lib \
  shell32.lib
```

```
conlibs: libc.lib ntdll.lib kernel32.lib
guilibs: libc.lib ntdll.lib kernel32.lib user32.lib gdi32.lib \
  winspool.lib comdlg32.lib
```

Most of the MFC samples link without error using the NTWIN32.MAK definitions of CONLIBS and GUILIBS, but will warn that WINSPOOL.LIB was not used. About half the samples do not need COMDLG32.LIB either. The CTRLTEST, MINSVR, MINSVRMI, OCLIENT, OSERVER, and TEMPLDEF examples require libraries not in the NTWIN32.MAK definition of GUILIBS.

Additional reference words: 3.10 3.1

PRB: GP Fault in OS/2 Subsystem

Article ID: Q92508

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A general protection fault (GP fault) can occur in the OS/2 subsystem if you have a corrupted registry entry for the subsystem.

A workaround for this is to start REGEDIT and edit HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft. Select OS/2 Subsystem for NT and press the DEL key. Now restart NT. Just logging off isn't sufficient; you must reboot. The next time you try to start an OS/2 application, the OS/2 server will rebuild the registry item you deleted.

SYMPTOMS

An application that causes a GP fault in the OS/2 subsystem displays the following error information:

General Protection error occurred
An OS/2 program caused a protection violation.
The program will be terminated.

Additional reference words: 3.10 3.1 GP-fault

INF: Cross-Platform Development Under Windows NT
Article ID: Q93213

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

There are currently three hardware platforms for which Win32 applications can be written; they are the Intel x86 (386 and higher), MIPS, and the DEC Alpha. How does one support development across these platforms?

Binary compatibility across these hardware platforms is not a viable alternative. Therefore, Win32 offers source compatibility. This means that developers may create versions of their applications for each CPU with simply a re-compile.

However, cross-compilation tools have not been written at this time. Thus, to develop separate executables for each hardware platform, developers must compile them within that hardware platform.

It is important not to confuse source compatibility across hardware platforms (x86, MIPS, Alpha) with binary compatibility across application execution environments (Win32, Win16). Win32 does support binary compatibility between different application execution environments upon the same hardware platform, through emulation. For example, a Win16 executable image (x86) can be run without modification on an x86 Windows NT machine, and with the support of the Win32s DLLs and VxDs, x86 Win32 executable images may be run on an x86 Windows 3.1 machine. Currently, development and debugging options across environments are limited during Windows NT early stages.

Additional reference words: 3.10

PRB: Internal Compiler Error msc1.cpp, Line 555
Article ID: Q93217

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

PROBLEM ID: SDK9212001

SYMPTOMS

The following error is given by the Preliminary SDK compiler (CL386.EXE):

Internal Compiler Error msc1.cpp, line 555

CAUSE

This error may indicate that the compiler binaries are corrupted.

RESOLUTION

If you encounter this error, compare the compiler binary files with those on the CD, to verify that file corruption is not the cause.

Additional reference words: 3.10 3.1

PRB: LINK32.EXE(): Extended Error - File Not Found
Article ID: Q93291

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

During linking, the following error occurs:

LINK32.EXE(): Extended error - File not Found.

CAUSE

This error may be caused by the TMP environment variable failing to point to an existing directory.

STATUS

Microsoft has confirmed this to be a problem with LINK32.EXE. We are researching this problem and will post more information here as it becomes available.

Additional reference words: 3.10 3.1

INF: Using the -ROM Linker Switch

Article ID: Q96014

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The linker (LINK32.EXE) included with the Win32 Software Development Kit (SDK) supports a -ROM switch for creating ROM images. When this switch is used, the executable header information created is different from a standard executable. For MIPS, the fixups are also different.

The ROM code and data points can be specified using the -BASE linker switch. The load address for the code and data can also be specified. These addresses don't need to be contiguous.

Control is transferred to the program through the ENTRYPOINT address in the header. It is up to the boot loader (or your own code) to switch into protected mode.

Additional reference words: 3.10 3.1

INF: Win32 .DEF File Usage in Applications and DLLs
Article ID: Q96374

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Certain aspects of Win32 .DEF file usage are the same as Win16 .DEF file usage; however, there are a number of important changes. Most of this information can be found under the Microsoft Library Manager (LIB) help in the help file "Compiler Tools Help," or in the help file "Building Applications and DLLs."

The .DEF files are not necessary to build an application (because callbacks don't need to be exported), but they can be used. However, the linker will not directly accept a .DEF file. The .DEF file is passed to the Library Manager. The Library Manager creates an export module (.EXP file), which needs to be linked with the application. Due to the nature of this step, it is necessary to include at least one export in the .DEF file.

Certain changes can be made without the .DEF file. If you need to set section attributes, you can use the linker -SECTION option. If you need to embed a description or a version string, you can use #pragma comment in the source files. Note that there currently is no support for the STUB statement either via the command line or via a .DEF file. The stub is hard-coded into the linker. This functionality is being considered for inclusion in either the final release of the Win32 Software Development Kit (SDK) or in Microsoft Visual C++ for Windows NT.

In Win32 development, a .DEF file is primarily used to export functions from a dynamic-link library (DLL). Again, the linker will not accept the .DEF file. Use the Library Manager to create the .EXP file to be linked with the DLL and to create an import library (.LIB file), which will be linked with any application or DLL that wants to use the DLL functions.

More Information:

The process of using a .DEF file to export functions requires that the EXPORTS listed in the .DEF file exactly match their functions' decorated names. The possible types of decorated names include:

1. `__cdecl`, where names are decorated with an underscore prefix
2. `__stdcall`, where names are decorated with an underscore prefix and an `@<number>` suffix, where `<number>` is the number of bytes in the argument list
3. C++, where the decorated name can be looked up in the output of `link32 -dump -symbols`

The Library Manager handles the name decoration for you if you pass it the object files on the command line. There is a special pass during which the Library Manager takes the undecorated name from the .DEF file, searches the object files, and provides the decorated name in the export module generated.

Note: For C++ overloaded functions or if two classes contain member functions that have the same name, the librarian cannot determine which decorated name to use and will issue an error. In this case, the fully decorated name must be provided in the .DEF file. There is no way to specify a scope resolution operator in the .DEF file.

For a sample makefile, see the SELECT or SPINCUBE sample.

Additional reference words: 3.10

INF: Win32 SDK Sample Build Warnings

Article ID: Q102114

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

Microsoft's goal was that the Win32 Software Development Kit (SDK) samples would compile cleanly at warning level 3 (-W3). However, due to time constraints, there are warnings that occur when building two of the samples on both x86 and MIPS. These warnings are as follows:

MSTOOLS\SAMPLES\SDKTOOLS\ANIEDIT

```
anicmd.c(768) : warning C4047: 'argument' : 'long ' differs in levels
of indirection from 'void *'
anicmd.c(768) : warning C4024: 'SendMessageA' : different types for
formal and actual parameter 4
```

MSTOOLS\SAMPLES\SNMP\SNMPUTIL

```
snmputil.c(463) : warning C4101: 'requestId' : unreferenced local
variable
```

Note that these warnings do not prevent the sample applications from running correctly, and therefore they can be ignored.

One error you may encounter is the following:

Q_A\SAMPLES\SHAREMEM

```
sharemem.cpp(204) : error C2106 : "=" : left operand must be l-value
```

This error is encountered if you rename SHAREMEM.C to SHAREMEM.CPP.

The following changes should be made to avoid these warnings:

```
snmputil.c(79) : delete line
```

```
    AsnInteger requestId;
```

```
anicmd.c(768) : typecast 4th parameter of SendMessage()
```

```
    SendMessage( hwndCaller, AIM_SETCHILAPP, 0, (LPARAM) hwnd );
```

```
sharemem.cpp(204) : typecast the right-hand side of the equation
```

```
    MapView = (LONG *) MapViewOfFile (...);
```

Additional reference words: 3.10

INF: LINK32 Implements New Switch: -adjust:#
Article ID: Q102115

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The version of LINK32 that is contained in the retail version of the Win32 Software Development Kit (SDK) implements a new switch, -adjust:#.

Previous beta linkers had a hard-coded limit for the number of iterations that are performed to do end-of-page code adjustments. The number of iterations can now be adjusted with this switch.

MORE INFORMATION

=====

This switch should be used when the following error is encountered:

error 101: cannot adjust code

Start with -adjust:20 and raise the number until the link is successful.

This error has been known to occur when linking large objects under MIPS. The R4000 and R4400 chips force us check that certain sequences of instructions do not occur at a page boundary. These sequences cause the system to take a page fault on crossing the page boundary, which the OS cannot resolve (however, this is not a problem with Windows NT). To prevent this from happening, the linker searches for these sequences in each code section in each object and moves the section up a DWORD at a time until there are no problems or until the number of maximum iterations is reached.

Additional reference words: 3.10

INF: Size Comparison of 32-Bit and 16-Bit x86 Applications
Article ID: Q97765

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

It is expected that a 32-bit version of an x86 application (console or GUI) will be larger than the 16-bit version. Much of this difference is due to the flat memory-model addressing of Windows NT. For each instruction, note that the opcodes have not changed in size, but the addresses have been widened to 32 bits.

In addition, the EXE format under Windows NT (the PE format) is optimized for paging; EXEs are demand-loaded and totally mappable. This leads to some internal fragmentation because protection boundaries must fall on sector boundaries within the EXE file.

The MIPS (or any RISC) version of a Win32 application typically will be larger and require more memory than its x86 counterpart.

Additional reference words: 3.10

INF: CTRL+C Exception Handling Under WinDbg

Article ID: Q97858

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

An exception is raised for CTRL+C only if the process is being debugged. The purpose is to make it convenient for the debugger to catch CTRL+C in console applications. For the purposes of this article, the debugger is assumed to be WinDbg.

When the console server detects a CTRL+C, it uses `CreateRemoteThread()` to create a thread in the client process to handle the event. This new thread then raises an exception IF AND ONLY IF the process is being debugged. At this point, the debugger either handles the exception or it continues the exception unhandled.

The "gh" command marks the exception as having been handled and continues the execution. The application does not notice the CTRL+C, with one exception: CTRL+C causes alertable waits to terminate. This is most noticeable when executing:

```
while( (c = getchar()) != EOF ) - or - while( gets(s) )
```

It is not possible to get the debugger to stop the wait from terminating.

The "gn" command marks an exception as unhandled and continues the execution. The handler list for the application is searched, as documented for `SetConsoleCtrlHandler()`. The handler is executed in the thread created by the console server.

After the exception is handled, the thread created to handle the event terminates. The debugger will not continue to execute the application if Go On Thread Termination is not enabled (from the Options menu, choose Debug, and select the Go On Thread Termination check box). The thread and process status indicate that the application is stopped at a debug event. As soon as the debugger is given a go command, the dead thread disappears and the application continues execution.

More Information:

There are three cases where CTRL+C doesn't cause the program to stop executing (instead it causes a "page down"):

1. When CTRL+C is already being handled.
2. When the debugger is in the foreground and a source window has the focus (both must be true).
3. When the CTRL+C exception is disabled (through the Debugger

Exceptions dialog box).

This follows the convention of the WordStar/Turbo C/Turbo Pascal editor commands.

Additional reference words: 3.10

INF: Debugging DLLs Using WinDbg

Article ID: Q97908

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

This article describes the process of debugging dynamic-link libraries (DLLs) under WinDbg. As a further example, debugging File Manager extensions is discussed in the "More Information" section in this article.

The application and the DLL must be built with certain compiler and linker switches so that debugging information is included. These switches can be found in the \$(cdebug) and \$(ldebug) macros, respectively, which are defined in NTWIN32.MAK.

Note: It is important to disable optimization with -Od or locals will not be available in the locals window and line numbers may not match the source.

The application is loaded into WinDbg either by specifying "windbg <filename>" on the command prompt or by starting WinDbg from the program group and specifying <filename> in the Program Open dialog box (from the Program menu, choose Open). Note that <filename> is the name of the application, not the DLL. It is not necessary to specify the name of the DLL to be debugged.

The DLL is loaded either when execution of the application begins or dynamically through a call to LoadLibrary(). In the first case, simply press F8 to begin execution. All DLLs and symbolic information are loaded. To trace through the DLL code, breakpoints can be set in the DLL using a variety of methods:

- From the Debug menu, choose Breakpoints. The dialog box is Program Open.

-or-

- Open the source file and use F9 or the "hand" button on the toolbar.

-or-

- Go to the Command window and type:

```
bp[#] <Options>
```

<Options>:

addr	break at address
@line	break at line

In the case that the DLL is dynamically loaded, pressing F8 causes all other DLLs and symbolic information to load. The same methods described above can be used to set breakpoints; however, the user will get a dialog box indicating that the breakpoint was not instantiated. After the call to LoadLibrary() has been executed, all breakpoints are instantiated (it is possible to note the color change if the DLL source window is open) and will behave as expected.

More Information:

To set a breakpoint in a DLL that is not loaded, specify the context when setting the breakpoint. The syntax for a context specifier is:

```
{proc, module, exe}addr
```

-or-

```
{proc, module, exe}@line
```

Example: {func, module.c, app.exe}0x50987. The first two parameters are optional, so {,,app.exe}0x50987 or {,,app.exe}func could be used instead.

For example, assume that we are trying to debug a File Manager extension that has been built with full debugging information. The procedure to debug the extension is as follows:

1. Open a Command window.
2. Start WinDbg WINFILE.
3. Set a breakpoint on FmExtensionProc().
4. At the Command window, type "g" and press ENTER. The debugger will continue executing the program from the point where it stopped (which could be from the beginning, at the breakpoint, and so on).

WinDbg will start WINFILE and when FmExtensionProc() is executed, WinDbg will break into the WINFILE process.

Additional reference words: 3.10

INF: Debugging Console Apps Using Redirection
Article ID: Q102351

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

To redirect the standard input (STDIN) for a console application named APP.EXE from a file named INPUT.TXT, the following syntax is used:

```
app < input.txt
```

However, the following syntax will not work when attempting to debug this application with STDIN redirected:

```
windbg app < input.txt
```

To debug the application as desired, use

```
windbg cmd /c "app < input.txt"
```

which will allow windbg to debug whatever goes on in the cmd window. A dialog box will be displayed that says "No symbolic Info for Debuggee." This message refers to CMD.EXE; dismiss this dialog box. When the child process (APP.EXE) is started, the command window will read "Stopped at program entry point." To continue, type "g" at the command window. Note that APP.EXE will begin executing, then you can open the source file and set breakpoints.

This technique is also useful when debugging an application that behaves differently when run with a debugger than it does when it is run in the command window.

Additional reference words: 3.10

PRB: Building POSIX Applications Under the March Beta
Article ID: Q97927

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

To build POSIX applications with the March beta release of the Win32 SDK for Windows NT, the reference to KERNEL32.LIB must be removed from NTWIN32.MAK, and the SETNVPSX.BAT file in the \MSTOOLS\POSIX directory must be run in order for the build environment variables to be set up correctly.

Additional reference words: 3.10

INF:**Article ID: Q98287**

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The code below fails with error C2440:

```
'return' : cannot conver from 'const char *' to 'char *':
```

Code

```
char * test () const { return          &chArray[0]; }
```

A member function is const and this member function receives a const pointer, which means that all members of the class are const; thus, the error is returned correctly.

The characters in the array are also const, and therefore the type of a pointer to the array is "const char *". If you want the member function to be const, you need to make the return type const as well.

Additional reference words: 3.10

INF: Watching Local Variables That Are Also Globally Declared
Article ID: Q98288

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Consider the following program:

```
int x = 1;
int y = 2;

main() {
    int x = 2;
    x++;
    y++;
}
```

Before you step into main, if you set watchpoints on x and y, the Watch window will display a value for y but for x will say "Expression cannot be evaluated." To see the value for x, use ::x and x will evaluate to the local x in main once you've stepped into main.

More Information:

When debugging an application, the X86 C++ evaluator is loaded. Given this, you can use the scope resolution operator in a watch statement to view a hidden global variable. Without the use of the scope resolution operator, there is no way (short of watching it in a memory window) to watch a hidden global variable.

Additional reference words: 3.10

INF: Migrating Windows NT Program Groups and the Desktop
Article ID: Q105298

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

There are a variety of ways of migrating groups under Windows NT. All of the desktop information is located in the registry. By selectively saving and restoring particular keys, you may restore your program groups and desktop.

Note that modifying the registry is a potentially risky operation; data loss is possible. You may want to back up your profile, which is located in <SYSTEMROOT>\SYSTEM32\CONFIG and is usually a combination of the username, some digits, and no extension. Your profile is locked by the system when you are logged in to that account, however, so you must copy it from another account or from MS-DOS.

The following is the procedure to save the desktop information:

1. Make sure you have Administrative access.
2. Run REGEDT32.EXE.
3. Select HKEY_CURRENT_USER\Program Groups.
4. From the Registry menu, choose Save Key.
5. Save the key (save as PROGRAM.REG).
6. Repeat steps 3-5 to save the following key:

```
HKEY_CURRENT_USER\  
Software\  
Microsoft\  
Windows NT\  
Current Version\  
Program Manager\  
Groups  
(save as GROUPS.REG)
```

8. Repeat steps 3-5 to save the following key:

```
HKEY_CURRENT_USER\Control Panel  
(save as CTRLPNL.REG)
```

PROGRAM.REG and GROUPS.REG have the information necessary to restore the program groups, and CTRLPNL.REG contains the information necessary to restore the desktop.

Procedure to restore registry keys:

1. Back up your profile.
2. Run REGEDT32.EXE.
3. Select the key you want to restore (see steps 3, 6, and 7 above).
4. Write down the exact name, spelling, and location of the key.
5. From the Edit menu, choose Delete and verify deletion of the key.
6. Select the parent key of the key you just deleted.
7. From the Edit menu, choose Add Key and type in the exact name of the deleted key. Don't worry about the Class field. Choose OK. The key should reappear in its position, but it will be empty.
8. Select the key that you just created.
9. From the Registry menu, choose Restore.
10. Find and select the file in which you saved the old key (for example, PROGRAM.REG). The saved key will be restored in the position that you selected.
11. Repeat for all keys you want to restore. When you are finished, exit REGEDT32 and log off. When you log back on, the groups and desktop should be restored.

Additional reference words: 3.10

INF: Conforming to ANSI C Standards

Article ID: Q98841

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Names starting with a single underscore and an uppercase letter, or names beginning with a double underscore and a lowercase letter, can be used in a "conforming" C program translated by a "conforming" implementation; these names, and the meaning and syntax associated with each name, are reserved by the implementation. ANSI C also specifies the requirements for a "strictly conforming" program, in which these constructions are not permitted.

Other compilers may ignore such names that exist in the implementation space, or produce any desired behavior, and the program can still be defined as conforming.

The Microsoft C compiler does not have a strictly conforming mode of operation. As a result, if a developer uses another vendor's compiler, which is a strictly conforming implementation, the Microsoft headers will not work. In this case, the vendor of the strictly conforming compiler must provide the appropriate headers for the system.

Because the standard headers are part of the implementation, there is no requirement for the header mechanisms provided by one implementation to be interoperable with the mechanisms of another. Another implementation MUST provide its own compatible set of standard headers and their associated inclusion mechanisms.

Additional reference words: 3.10

PRB: Default Section Alignment Is 0x10000 (64K) by Default
Article ID: Q98889

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

The help for the Win32 Software Development Kit (SDK) linker indicates that the default section alignment is set to 0x1000 (4K). By using `link32 -dump -header` on an executable, it is possible to see that the alignment is set to 0x10000 (64K) by default.

RESOLUTION

To manually set the alignment to 4K, use `-align:0x1000`. In versions of the Beta Win32 SDK that are later than the March release, the `NTWIN32.MAK` file uses the `-align:` switch to force alignment to 4K.

STATUS

Microsoft has confirmed this to be a problem with the March Win32 SDK linker (LINK32).

More Information:

Although the application loads with 64K alignment, the extra pages of supposedly committed memory are not really committed. If you analyze them using `VirtualQuery()`, you will find that they are marked as `MEM_COMMIT` and `PAGE_NO_ACCESS`. Although this typically means that the pages are committed and have backing store somewhere (most likely the system pagefile), they actually do not.

In the next release, these extra committed pages will be marked as `MEM_RESERVE` and will occupy nothing more than address space in the process. Therefore, although it appears that the application takes a large amount of memory to load, only the pages that are needed are committed and no extra backing store is allocated.

Additional reference words: 3.10 3.1

PRB: Cannot Compile from Win32 SDK M Editor (MEP.EXE)
Article ID: Q98918

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

The Win32 Software Development Kit (SDK) M Editor compile function uses the correct extmake line and spawns the compiler correctly, but does not produce the desired result. The terminating beep is issued, but no messages appear in the <compile> pseudo file and no object file is produced.

STATUS

Microsoft has confirmed this to be a problem in the Win32 SDK Microsoft Editor.

Additional reference words: 3.10 3.1

INF: UNICODE and _UNICODE Needed to Compile for Unicode
Article ID: Q99359

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

To compile code for Unicode, you need to #define UNICODE for the Win32 header files and #define _UNICODE for the C run time. These #defines must appear before the #include <WINDOWS.H> and any #included C run-time headers. The leading underscore indicates ANSI-deviance from the C standard. Because the Windows header files are not part of any standard, this is allowable.

Additional reference words: 3.10

INF: Specifying Filenames Under the POSIX Subsystem
Article ID: Q99361

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

When specifying filenames under POSIX, use

```
//c/subdir/executable.exe
```

to specify c:\SUBDIR\EXECUTABLE.EXE. If you fail to use this format, you will receive ENAMETOOLONG as the errno.

Additional reference words: 3.10

PRB: WM_QUERYOPEN Incorrectly Prototyped in WINDOWSX.H
Article ID: Q99362

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

SYMPTOM

When MAKELRESULT is used with the WM_QUERYOPEN message cracker, a compiler error is issued.

CAUSE

In the March Beta release of the Win32 Software Development Kit (SDK), the message cracker for WM_QUERYOPEN in WINDOWSX.H is incorrect. The second parameter to MAKELRESULT is missing, thus causing the compiler error when it is used. The second parameter should be 0L.

RESOLUTION

Developers using this message cracker should make the change locally in WINDOWSX.H until this file is updated to reflect this change.

Additional reference words: 3.10

INF: Using SetThreadLocale() for Language Resources
Article ID: Q99392

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

Under Windows NT, each resource loading application programming interface (API) is based on the thread's locale. Each thread has a locale--usually the default system locale.

You can change the thread locale by calling SetThreadLocale(). To obtain the language resource you want, just set the thread locale to the locale you want, then call the normal resource loading API.

Additional reference words: 3.10

INF: Compile Errors Caused by Missing Option -D_X86_
Article ID: Q99516

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

When compiling Windows applications on an Intel x86 system, the following errors are generated if the symbol `_X86_` is not defined:

```
C:\MSTOOLS\h\winnt.h(1235) : error C2061: syntax error : identifier
'PCONTEXT'

C:\MSTOOLS\h\winnt.h(1236) : error C2143: syntax error : missing ';'
before '}'

C:\MSTOOLS\h\winbase.h(615) : error C2282: 'PCONTEXT' is followed by
'LPCONTEXT' (missing ',')

C:\MSTOOLS\h\winbase.h(617) : error C2282: 'PEXCEPTION_POINTERS' is
followed by 'LPEXCEPTION_POINTERS' (missing ',')

C:\MSTOOLS\h\winbase.h(1409) : error C2061: syntax error : identifier
'lpContext'

C:\MSTOOLS\h\winbase.h(1416) : error C2059: syntax error : '*'

C:\MSTOOLS\h\winbase.h(1416) : error C2061: syntax error : identifier
'lpContext'
```

Certain sections of WINNT.H and WINBASE.H are conditionally compiled with:

```
#ifdef _X86_
```

Therefore, failing to compile with the `-D_X86_` switch causes these errors to occur.

More Information:

For recommendations regarding which compile and link options to use, check the makefiles included with the samples. These makefiles use the macros defined (and described) in `\MSTOOLS\H\NTWIN32.MAK`.

Additional reference words: 3.10

PRB: Running Early Apps Results in Error w/ RtlExAllocateHeap
Article ID: Q103241

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SYMPTOMS

=====

The following error is received

The procedure entry point RtlExAllocateHeap could not be located in the dynamic link library ntdll.dll.

when running applications on Windows NT version 3.1 that were built with a version of the Win32 SDK earlier than the .404 (March) release.

CAUSE

=====

Applications built with the .404 (March) SDK or later should have no problems running on Windows NT version 3.1. Applications built with earlier SDKs are not guaranteed to run on the final release Windows NT, even if they ran on the .404 release of Windows NT.

WORKAROUND

=====

The best way to work around the problem is to completely rebuild the application with the latest tools. If this is not possible, the other alternative is to patch the old executable with B2FIX.EXE, which can be found in Library 1 of the MSLNG32 forum. The original intent of this program is to patch Visual C++ 1.0 so that it can be used under Windows NT.

MORE INFORMATION

=====

Note that not all of the March SDK tools were built with the March SDK. In particular, running the March SDK cl386 compiler will cause this error. It is necessary to use the final SDK update or Visual C++ for Windows NT to create 32-bit applications on Windows NT.

Additional reference words: 3.10

INF: Undocumented Warning C4509

Article ID: Q103242

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

The following warning was added to the Win32 SDK compiler and the Visual C++ for Windows NT compiler:

```
warning C4509: nonstandard extension used: '<function>' uses SEH and
'<variable>' has destructor
```

The warning was added in anticipation of a future compiler that will support C++ exception handling (C++ EH). At that time, it will not be possible to mix C++ EH and structured exception handling (SEH). Because destructors will involve C++ EH so that they can be called in an exception unwind, it will not be possible to have a local object in a function that uses SEH.

Currently, we do not destruct objects in an exception unwind.

MORE INFORMATION

=====

Code that produces the warning described above, where <function> is MyFunction and the variable is mc, resembles the following:

```
struct MyClass {
    MyClass();
    ~MyClass();
};

void MyFunction( )
{
    MyClass mc;

    try {
        //...
    }

    finally
    {
        //...
    }
}
```

Additional reference words: 3.10

INF: Example of Importing Functions

Article ID: Q103244

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

The linker does not take a .DEF file directly. To use EXPORTS, it is standard to build an export module (.EXP) to link with the dynamic-link library (DLL), and to use an import library (.LIB) to link with any application or DLL that uses this DLL.

There are alternatives to an import library. One alternative is to use LoadLibrary() and GetProcAddress() to call an exported DLL function. Another alternative is to build an import module (.IMP) using the steps described below:

1. Create the .DEF file. The following is based on the SELECT example. Note that it is necessary to decorate the names and include one dummy export. The __stdcall functions have an appended "@<number>", where <number> is the number of bytes in the parameter list for the function. The linker automatically handles the underscore, which is prepended to __cdecl and __stdcall function names.

```
NAME                Demo

EXPORTS
    DemoWndProc@12

IMPORTS
    select.StartSelection@16
    select.UpdateSelection@16
    select.EndSelection@8
    select.ClearSelection@12
```

2. Build the .IMP file:

```
$(PROJ).imp: $(PROJ).def
    $(implib) -machine:$(CPU) \
    -def:$(PROJ).def \
    -out:$(PROJ).lib
```

3. Link the .IMP file into the application:

```
$(PROJ).exe: $(PROJ).obj $(PROJ).rbj $(PROJ).def $(PROJ).imp
    $(link) $(linkdebug) $(guiflags) \
    -out:$(PROJ).exe $(PROJ).obj $(PROJ).rbj \
    $(PROJ).imp $(guilibsdll)
```

MORE INFORMATION

=====

If building a DLL, another interesting technique is to use forwarders. For example, the following

```
LIBRARY test
```

```
EXPORTS
```

```
    MyFunc = prvtddl.DllFunc
```

will expose MyFunc() as an export for TEST.DLL. However, the loader will actually fix up the reference at load time to point to DllFunc in PRVTDLL.DLL. This method involves no additional code.

Another way to provide a forwarder is to write a stub function, which you do export, that forwards the reference for you. On DLL PROCESS_ATTACH, do LoadLibrary(prvtddl), then GetProcAddress() on DllFunc. In your forwarder function, just call through with the arguments passed to you.

Additional reference words: 3.10

PRB: Error in Win32 SDK Install Program MANUAL.BAT
Article ID: Q103245

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SYMPTOMS

=====

When installing the Win32 SDK with an unsupported CD-ROM, Microsoft recommends booting MS-DOS and running MANUAL.BAT. This results in the following error:

```
invalid switch - /i
```

WORKAROUND

=====

One workaround is to copy the MANUAL.BAT file to the hard disk and remove the /i switches on the xcopy command lines. For example, if you copied the file to C:\MANUAL.BAT and wanted to install the Win32 SDK with the unsupported CD-ROM to drive c, the next step would be to switch back to the CD-ROM and run C:\MANUAL.BAT C:.

In addition, a corrected copy of the MANUAL.BAT file was uploaded to Library 8 of the MSWIN32 forum on CompuServe:

```
SETUP.ZIP/Bin Bytes: 10507, Count: 0, 02-Sep-93  
Last:**Never**
```

```
Title : Manual Bat Installation Helper Tools  
Keywords: SETUPSDK MANUAL SETUP FIX TOOL
```

This file contains: MANUAL.BAT & SETUPSDK.EXE

MANUAL.BAT is the fixed version of same-named file found on the Win32 SDK CD_ROM. This is the same file as found in MANUAL.ZIP as found in this forum.

The SetupSDK tool will allow you to selectively create the various Program Manager groups and environment variables as are normally created using the "Graphical Installation" method. This tool can also be used to create these groups and variables for different user accounts on your local machine. This tool runs under Windows NT only.

Additional reference words: 3.10

PRB: Destructor for Class in a DLL Called Twice

Article ID: Q103860

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SYMPTOMS

=====

The application has the following properties and generates an access violation at exit:

- There is a C++ class in a dynamic-link library (DLL)
- Objects of this class type use dynamic memory
- The application was built using the SDK tools

CAUSE

=====

Under the following conditions, the destructor for the class in the DLL will be called twice, and therefore the memory will be freed twice, causing the access violation.

RESOLUTION

=====

This problem does not occur when using Visual C++ for Windows NT.

MORE INFORMATION

=====

This problem can be easily demonstrated by adding a CString to the DLLTrace sample program.

Additional reference words: 3.10

INF: Choosing the Debugger That the System Will Spawn
Article ID: Q103861

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

With Windows NT version 3.1, it is possible to have the system spawn a debugger whenever an application faults. The capability is controlled by the following Registry key:

```
HKEY_LOCAL_MACHINE\  
    SOFTWARE\  
        Microsoft\  
            Windows NT\  
                CurrentVersion\  
                    AeDebug
```

This key contains the following values:

```
Auto  
Debugger
```

If the value of Auto is set to "0" (zero), then the system will generate a pop-up window, and if the user chooses Cancel, spawn the debugger that is specified in the Debugger value. If the value of Auto is set to "1", then the system will automatically spawn the debugger that is specified in the Debugger value.

After installing Windows NT 3.1, the Debugger value is set to DRWTSN32 -p %ld -e %ld -g and the Auto value is set to 1.

If the Win32 SDK is installed, then the Debugger value is changed to <MSTOOLS>\BIN\WINDBG -p %ld -e %ld and the Auto value is set to 0.

MORE INFORMATION

=====

The DRWTSN32 debugger is new and is a post-mortem debugger similar in functionality to the Windows 3.1 Dr. Watson program. DRWTSN32 generates a log file containing fault information about the offending application. The following data is generated in the DRWTSN32.LOG file:

- Exception information (exception number and name)
- System information (machine name, user name, OS version, and so forth)
- Task list
- State dump for each thread (register dump, disassembly, stack walk, symbol table)

A record of each application error is recorded in the application

event log. The application error data for each crash is stored in a log file named DRWTSN32.LOG, which by default is placed in your Windows directory.

Additional reference words: 3.10

INF: Symbolic Information for System DLLs

Article ID: Q103862

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

The debugging information for the system dynamic-link libraries (DLLs) is contained separately in files with a .DBG extension. The Win32 SDK setup, SETUPSDK, will install the following .DBG files by default in <SYSTEMROOT>\SYMBOLS\DLL:

ADVAPI32.DBG	OLECLI32.DBG
COMDLG32.DBG	OLESVR32.DBG
CRTDLL.DBG	RASAPI32.DBG
DLCAPI.DBG	RPCNS4.DBG
GDI32.DBG	RPCRT4.DBG
INETMIB1.DBG	SHELL32.DBG
KERNEL32.DBG	USER32.DBG
LMMIB2.DBG	VDMDBG.DBG
LZ32.DBG	VERSION.DBG
MGMTAPI.DBG	WIN32SPL.DBG
MPR.DBG	WINMM.DBG
NDDEAPI.DBG	WINSTRM.DBG
NETAPI32.DBG	WSOCK32.DBG
NTDLL.DBG	

Note that these files are not installed by the alternative Win32 SDK install method, MANUAL.BAT. Therefore, WinDbg will warn that symbol information cannot be found for each of the system DLLs called by the debuggee.

These .DBG files can be manually installed by copying them from the SDK CD. For x86, they are located in \SUPPORT\DEBUG\I386\SYMBOLS\DLL. For MIPS, they are located in \SUPPORT\DEBUG\MIPS\SYMBOLS\DLL. Note that there are more than 200 .DBG files in each of these directories.

MORE INFORMATION

=====

There are also debugging versions of the system DLLs that can be installed by using SWITCH.BAT, which is located on the CD in \SUPPORT\DEBUGDLL. Refer to page 11 of the "Getting Started" manual and the batch file itself for more information.

Additional reference words: 3.10

INF: Cannot Load <exe> Because NTVDM Is Already Running
Article ID: Q103863

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

The version of WinDbg that is included in the Win32 SDK version 3.1 can debug 16-bit applications running on Windows NT, under the Win16 VDM (virtual DOS machine), NTVDM.

It is a requirement that NTVDM cannot already be running, and therefore when debugging a 16-bit application, no other 16-bit applications can be running. If NTVDM is running, you will get the following error message:

Cannot load <exe> because NTVDM is already running

To terminate NTVDM, run PView, select NTVDM, and choose "Kill Process." Note that there may be two NTVDM processes. The one that you want to terminate has one thread for each Win16 application (plus a few more).

The Windows NT WinLogon is set up to automatically start WoWExec, which automatically starts the Win16 VDM. This behavior can be changed by removing WoWExec from:

```
HKEY_LOCAL_MACHINE\  
    Software\  
        Microsoft\  
            Windows NT\  
                CurrentVersion\  
                    Winlogon\  
                        Shell
```

Additional reference words: 3.10

PRB: Win32 SDK and VC++ NT Help Files Are Incompatible
Article ID: Q104378

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The Windows Help files that accompany the Win32 SDK (that is, API32WH.HLP) are incompatible with Windows Help files from the Visual C++ for Windows NT product.

SYMPTOMS

=====

Choosing the Search Plus button from within Visual C++ for Windows NT Windows Help with a copy of a help file from the Win32 SDK produces a "Missing or Old Index" error.

CAUSE

=====

The version of API32WH.HLP for Visual C++ for Windows NT is indexed with several other help files to make it easy to search for terms and keywords in several files at once. The Visual C++ Setup program writes the name of the new index file into the Windows NT registry, and this name is different than the name of the index file included with the Win32 SDK. If you subsequently open API32WH.HLP from the Win32 SDK and use the full-text search feature (the Find button), you will receive an error message about an old or missing index file.

RESOLUTION

=====

If you have previously installed the Win32 SDK for Windows NT, after installing Visual C++ for Windows NT, you should use only the version of the Win32 application programming interface (API) help file (API32WH.HLP) installed with Visual C++ for Windows NT. The Setup program installs this file by default in the \HELP subdirectory under the directory that you have chosen for Visual C++, or in a help subdirectory that you explicitly choose.

For more information about moving from the Win32 SDK to Visual C++ for Windows NT, see the MIGRATE.HLP help file.

Additional reference words: 3.10

INF: Development Tools Do Not Accept Unicode Text

Article ID: Q106065

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Neither the Win32 SDK tools or Visual C++ (VC++) support Unicode text. In fact, the C/C++ Language specification says that the source files are to be written in 7-bit ANSI.

For example, language-specific resources cannot be specified in Unicode in the .RC file because RC does not accept the Unicode text. Although the message compiler has flags for Unicode, the flags are not implemented.

To convert to and from Unicode text, use the UCONVERT utility included in your MSTOOLS\BIN directory. The source for UCONVERT is in MSTOOLS\SAMPLES\SDKTOOLS\UCONVERT.

The long term solution that Microsoft is working on are Resource Localization Tools and other methods that will allow the user to localize the strings in a GUI editor, running on the target machine.

Note that it is possible to specify Unicode escapes in L-quoted strings. The following is quoted from "Common Statement Parameters" in RC.HLP:

By default, the characters listed between the double quotation marks are ANSI characters and escape sequences are interpreted as byte escape sequences. If the string is preceded by the L prefix, the string is a wide-character string and escape sequences are interpreted as two-byte escape sequences that specify Unicode characters. If a double quotation mark is required in the text, you must include the double quotation mark twice or use the \" escape sequence.

Another alternative is to use user-defined resources and include a binary (Unicode) file.

Additional reference words: 3.10

INF: Viewing Globals Out of Context in WinDbg
Article ID: Q105583

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

When viewing global variables (either with the ? command or via the Watch window) and the variables go out of context, their values become:

```
CXX0017 Error: symbol not found
```

An example of this is when a common dialog box is open in the application. If you break into an application that is inside COMDLG32.DLL and try to do a ?gVar, where gVar is a global variable in the application, WinDbg will not find the symbol because the context is wrong. To view the value of gVar in MYAPP, use the following:

```
?{,,myapp}gVar
```

WinDbg will then have no trouble locating the symbolic information.

Additional reference words: 3.10

PRB: RC Does Not Support __DATE__ or __TIME__
Article ID: Q105584

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

The resource compiler (RC) included with the Win32 Software Development Kit (SDK) does not support the predefined symbols __DATE__ or __TIME__. Previous versions of the resource compiler accepted these symbols.

CAUSE

=====

It is not intended that the resource compiler support these symbols. Previous versions of RC used the C/C++ 7.0 preprocessor, which supported these symbols.

RESOLUTION

=====

If code that uses this symbol is in a header file, place the following around the individual statements that use them:

```
#ifdef RC_INVOKED
```

Additional reference words: 3.10

PRB: Unable to Freeze One Thread in WinDbg
Article ID: Q105585

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

The Notes section of the "Set thread" entry of the WinDbg Help file states:

In any of the above listed states, a thread may be frozen. If a thread is at a breakpoint, no threads in that process will run. If a thread is at a breakpoint and then frozen, it will still be considered stopped until the thread is continued (at which time it will still be considered frozen).

If you set a breakpoint in a multithreaded application and freeze a thread after the breakpoint is hit, all threads stop, as expected. If you then choose GO on the frozen thread, the message in the Command window is "Thread is still frozen," which seems fine but all other threads remain blocked until the thread is unfrozen.

CAUSE

=====

The implementation of ContinueDebugEvent() uses the thread that is being continued to do some of its work. If the thread is suspended, it cannot run; therefore, the ContinueDebugEvent operation is not finished until the thread is resumed.

STATUS

=====

Microsoft has confirmed this to be a limitation in WinDbg.

Additional reference words: 3.10

PRB: WinDbg FIND Dialog Box Slows Down the System
Article ID: Q105586

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

Leaving the WinDbg FIND dialog box displayed after searching for some string causes the CPU to go to 100 percent busy, making the system very slow.

RESOLUTION

=====

Choosing Cancel in the FIND dialog box allows the system to return to normal.

STATUS

=====

Microsoft has confirmed this to be a problem in WinDbg.

Additional reference words: 3.10

INF: Debugging the Win32 Subsystem

Article ID: Q105677

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The instructions on page 1-18 of Part II of the Win32 "Programmer's Guide" included with the Win32 Device Driver Kit (DDK) says to use NTSD -d -p -1 to attach to the Win32 subsystem process and enable debugging of its user-mode drivers. This results in the error:

```
NTSD: cannot debug PID -1
error = 5
```

To enable this procedure to work properly, change the GlobalFlag value under:

```
HKEY_LOCAL_MACHINE\
  SYSTEM\
    CurrentControlSet\
      Control\
        Session Manager
```

Remove the flag 0x00080000 from 0x211a0000 to make it 0x21120000. The 0x00080000 flag disables the ability to debug CSRSS.EXE (the client server run time subsystem), which is specified by the "-p -1" parameter.

It is also possible to debug CSRSS using "WinDbgRm -c -p-1" instead of NTSD. Make sure that WinDbgRm defaults to debugging using TLPIPE.DLL as its transport layer, then run "windbgm -c -p-1" on the debuggee.

On the debugger machine, make sure that CSRSS.EXE and any dynamic-link libraries (DLLs) that you are debugging in association with it are in the same directory, and run WinDbg. To set the transport DLL, choose Debug from the Options menu, choose Transport DLLs, and set the transport DLL to TLPIPE. Set the host name entries to be the machine name of the debuggee.

Additional reference words: 3.10

INF: Differences Between the Win32 SDK and 32-Bit VC++
Article ID: Q105679

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The following is a list of items included in the retail Win32 SDK that are not included in the 32-bit Edition of Visual C++ (Intel only):

Tools

WinDbg/WinDbgRM
Process Walker
Working Set Tuner (WST)
WinObj
Setup Toolkit
Microsoft Test
Software Compatibility Test (SCT)
API Profilers: CAP and WAP
Symedit
masm386
Font Editor

Documentation

LM API Reference (LMAPI.HLP)
SNMP Programmer's Reference (SNMP.TXT)
Generic Thunks (GENTHUNK.TXT)
File Formats (CUSTCNTL.TXT, ENHMETA.TXT, PE.TXT, RESFMT.TXT)

Samples

BNDBUF	MFEDIT
BOB (named EXITWIN in VC++)	MIDIMON
CDTEST	MINREC
CPL	MSGTABLE
DYNDLG	NTFONTS
GUIGREP	NTSD
LARGEINT	PDC
LOGGING	RESDLL
LOWPASS	REVERSE
MANDEL	SCRNSAVE
MAPI	SEMAPHOR
MAZELORD	SPINCUBE
MCITEST	WINNET

Other

- Device Driver Kit (DDK) headers and libraries
- Checked build of Windows NT
- RPC Toolkit
- POSIX headers and libraries
- Microsoft Foundation Classes (MFC) 1.0

The following is a list of items included in the 32-bit Edition of Visual C++ that are not included in the retail Win32 SDK (Intel only):

Tools

- Visual Workbench/AppStudio/Wizards
- bscmake
- Pharlap TNT DOS-Extender
- Spy++
- Source Profiler
- CodeView for Win32S

Samples

- BOUNCE
- CVTMAKE
- EXITWIN (named BOB in the SDK)

Other

- MFC 2.0

MORE INFORMATION

=====

The preliminary Win32 SDK contained the compiler tools, while the retail Win32 SDK does not. In addition, the preliminary Win32 DDK was available separately, while the retail DDK is bundled with the retail SDK.

The Win32 SDK has a separate "Win32s Programmer's Reference," while VC++ has the same chapters as part of the "Programming Techniques" manual.

There are tools whose names have changed and tools that are no longer needed. The SDK linker is LINK32, the VC++ linker is LINK. The SDK librarian is LIB32, the VC++ librarian is LIB. CVTRES existed in the SDK to convert the .RES file produced by RC so that it could be used by the linker, while VC++ has this functionality built into its linker. These changes may affect your makefiles.

For more information on switching from the Win32 SDK to 32-bit VC++, see the VC++ file MIGRATE.HLP.

Additional reference words: 3.10

INF: Listing the Named Shared Objects

Article ID: Q105764

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Included with the final retail product is an object viewer utility called WinObj that be used to list named objects, devices, dynamic-link libraries (DLLs), and so forth. To find objects such as pipes, memory, and semaphores, start WinObj and select the folder BaseNamedObjects.

Additional reference words: 3.10

INF: Additional Remote Debugging Requirement
Article ID: Q106066

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The printed and online documentation for remote debugging with WinDbgRm fail to mention one requirement. The binaries must be in the same drive and directory on both the target machine and the development machine.

WinDbg also expects to find the source files in the same directory in which the the binary file was built, but will browse for the source if it is not found in this location. WinDbg will automatically locate the source if the files are specified to the compiler with fully qualified paths.

Additional reference words: 3.10

PRB: Problems with the Microsoft Setup Toolkit
Article ID: Q106382

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

1. When the /zi option is used with the Win32 DSKLAYT2.EXE to provide compression, it causes an access violation.
2. The Win32 Setup Toolkit does not contain a setup bootstrapper to copy the needed setup files to a temp drive and run the Setup program. Setup runs from floppy disks.
3. Install programs for applications that may run on Win32s must be created with the 16-bit version of the Setup Toolkit if the installation program will install Win32s. However, the 16-bit DSKLAY2.EXE cannot read the version information in a Win32 binary.
4. The Win32 DSKLAYT.EXE only shows 8.3 names in the list box.

RESOLUTIONS

=====

1. The fix for this problem is available in the Alpha SDK Update. This product contains Alpha development tools as well as updates to the x86 and MIPS components.

Note that COMPRESS.EXE has been updated to use a better compression algorithm, and therefore /zi is no longer recommended for best compression. The option has been kept for compatibility reasons.

2. The bootstrapper is not necessary in a 32-bit environment. It is required for Windows because it is not possible to remove the floppy disk of a currently running Win16 application (the resources could not all be preloaded and locked). If you want to use a bootstrapper for compatibility, a 32-bit version is available on CompuServe.
3. If a Win32s installation is provided on a separate disk, the install program can be developed with the Win32 Setup Toolkit.
4. The program is actually a 16-bit program, and therefore it can display only the 8.3 name. Use 8.3 names for the source names and specify that the files be renamed (using the long names) when they are installed.

Additional reference words: 3.10

INF: Win32s Stacks Limited to 128K

Article ID: Q93547

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Win32 applications running under Win32s are limited to a maximum of 128K of stack space.

Additional reference words: 3.10

INF: Description of Win32s API

Article ID: Q83520

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The following is intended as an introduction to Win32s.

MORE INFORMATION

=====

General Overview

Win32s is an API (application programming interface) intended for development for both Windows 3.1 (hereafter referred to as Win16) and Win32 environment. The purpose of Win32s is to provide an API to developers, which will allow binary-compatible code across both environments.

Win32 consists of the Win16 API stretched to 32 bits, plus the addition of new API functions that offer new functionality. Win32s contains a subset of the Win32 API, function stubs for the Win32 API functions that Win16 will not support, and new API to support the universal thunk (UT). Among the new features gained from Win32 are structured exception handling (SEH), FP emulation, memory-mapped files, named shared memory, and sparse memory. ALL the Win32 API functions are included in Win32s; however, those that are not supported will return an error code. For details on which API are supported under Win32s, refer to the entries in the help file "Win32 API Reference."

When you call a Win32s API function, two options should be allowed:

- Option A: Your code should allow for a successful return from the function call.
- Option B: Your code should allow for an unsuccessful return from the function call.

For example, if the code is running under Windows 3.1 and the function call is made to one of the API functions in the subset supported on Windows 3.1, then the call returns successfully and option A is executed. If the call is made while running under the Windows NT environment, the call again returns successfully and option A should be executed. However, if running under Windows 3.1 and a Win32s API function is called that is unsupported by Windows 3.1 (this is one of the Win32s function stubs), then an error code is returned and option

B should be executed.

If, for example, option A were using a `CreateThread()` call, then option B would be alternative code, which would handle the task using a single-thread solution. The list of APIs that may return error codes on Windows 3.1 include:

- beziers
- enhanced metafiles
- mutex
- overlapped I/O
- paths
- semaphores
- threads
- Winnet APIs
- world coordinate transforms

Architecture

An application for Win32 using the Win32s toolkit will find some of its API functions serviced within a 32-bit VxD, and some of its API functions serviced by the 32-bit DLLs (dynamic-linked libraries). In general, "base" functionality will go to the VxD, and the Win16 API function equivalents will go to the DLLs.

Applications for Win32 cannot use Interrupt XX functionality; therefore, the Win32s VxD has Win32 entries for each Interrupt 21 and various other Interrupt XX (BIOS) calls. The Win32 memory management API functions are directly implemented in the Win32s VxD (32-bit memory management cannot be mapped to 16-bit reasonably) and the Interrupt XX equivalents are mapped back to the virtualized MS-DOS.

The Win32s DLLs "thunk" back to Win16 when an application for Win32 makes a call, so the Win32s DLL 32-bit parameters are copied from the 32-bit stack to a 16-bit stack, and the 16-bit entry point is called. Note that you are allocated Window handles only up to `0x0000FFFF`, as expected. However, do not assume this because it will not be true when the same binary is run under Win32.

There are other semantic difference between Windows 3.1 and Win32. Windows 3.1 will run applications for Win32 nonpreemptively in a single, shared address space, while Windows NT runs them preemptively in separate address spaces. It is therefore important that you test your application for Win32 on both Windows 3.1 and Win32.

If you need to call 16-bit functionality from 32-bit code, you may do this using universal thunks under Win32s, RPC, or other client-server techniques. DDE, OLE, the clipboard, metafiles, and bitmaps can be used between Win16 and Win32 applications on both Windows 3.1 and Windows NT. For a description of UT, see the "Win32s Programmer's Reference."

32-Bit Compatibility

Win32s offers 32-bit compatibility on Windows 3.1; therefore, it will

offer a speed improvement over Win16 applications running under Win32. However, the actual speed improvement varies with each application because it depends on how often you cross the thunk layer. Each call using a thunk is no more than 10 percent slower than a call not using a thunk.

Win32s on Windows 3.1 has 32-bit drivers (VxDs), some of which call to 16-bit file systems (MS-DOS in virtual mode), and some of which directly implement functionality (Win32s memory management). Win32s on Windows 3.1 also has a 32-bit preemptive kernel (Windows enhanced mode) and 16-bit graphics and windowing (GDI and USER).

Additional reference words: 3.10

INF: Use 16-Bit .FON Files for Cross-Platform Compatibility
Article ID: Q100487

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The AddFontResource() function installs a font resource in the GDI font table. Under Windows NT, the module can be a .FON file or a .FNT file. Under Windows 3.1, the module must be a .FON file. When using Win32s, AddFontResource() passes its argument to the Win16 AddFontResource, and therefore .FON files should be used for portability.

In addition, when running under Windows NT, the module can be either a 32-bit "portable executable" or a 16-bit .FON file. However, if the same Win32 executable is run under Win32s, the call to AddFontResource() fails if the *.FON is not in 16-bit format. Therefore, for compatibility across platforms, use 16-bit *.FON files. These can be created using the Windows 3.1 Software Development Kit (SDK).

Additional reference words: 3.10

INF: Support for Sleep() on Win32s
Article ID: Q100713

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The Win32 application programming interface (API) documentation indicates that Sleep() is supported on Win32s. It is important to note, however, that the behavior of Sleep() on Win32s is not the same as it is under Windows NT.

Under Win32s, Sleep() calls Yield(). The Windows version 3.1 Yield() function yields only if the message queue is empty; therefore, Sleep() cannot be relied on to do anything. Use a PeekMessage() loop to do idle time processing.

Additional reference words: 3.10

INF: Win32s Translated Pointers Guaranteed for 32K
Article ID: Q100833

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

Translated pointers are guaranteed to be valid only for 32K, rather than 64K, which selectors are usually limited to. This limitation is for performance reasons.

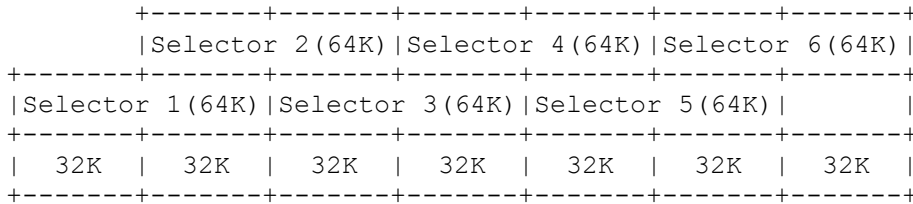
Selectors are tiled every 32K. A 0:32 pointer can be quickly translated into a 16:16 pointer, which will be valid for a minimum of 32K. In other words, the offset portion of the 16:16 pointer is not guaranteed to be 0 (zero) when translated. As a result, even though the translated selectors have a limit of 64K, the offset passed to the 16-bit side may be as large as 32K-1.

The alternative is to create a selector for each and every translation, which is very slow.

MORE INFORMATION

=====

For any given address, there are two selectors that point to it, but only one has a limit less than 32K:



Currently, 16-bit and 32-bit applications share the same global data space; therefore, it is possible to share a buffer of up to 64K in size with a far pointer or more than 64K with a huge pointer by doing the following:

1. Do a GlobalAlloc() on the 32-bit side.
2. Copy the data.
3. Send the handle to the 16-bit side.
4. Get a pointer to the data on the 16-bit side by using GlobalLock().

The translated pointer is valid until the memory is freed. In the future, however, 16-bit and 32-bit applications may not share the same global data space. Therefore, the recommendation is to use a buffer

that is passed to `UTRegister()`.

Additional reference words: 3.10

INF: Calling a Win32 DLL from a Windows 3.1 Application
Article ID: Q97785

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

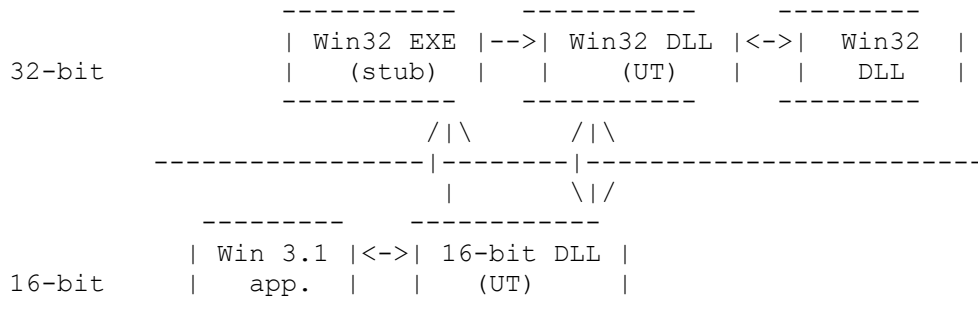
Summary:

A Windows version 3.1 application can call a Win32 dynamic-link library (DLL) under Win32s using Universal Thunks.

The following are required components (in addition to the Windows 3.1 application and the Win32 DLL):

- A 16-bit DLL that provides the same entry points as the Win32 DLL. This serves as the 16-bit end for the Universal Thunk. The programmer must also include code that will detect whether the 32-bit side is loaded.
- A Win32 DLL that sets up the Universal Thunk. This serves as the 32-bit end of the Universal Thunk. This DLL is supported only under Win32s.
- A Win32 EXE that loads the 32-bit DLL described above.

The following diagram illustrates how the pieces fit together:



MORE INFORMATION

=====

The load order is as follows: The Windows 3.1 application loads the 16-bit DLL. The 16-bit DLL checks to see whether the 32-bit side has been initialized. If it has not been initialized, then the DLL spawns the 32-bit EXE (stub), which then loads the 32-bit DLL that sets up the Universal Thunks with the 16-bit DLL. Once all of the components are loaded and initialized, when the Windows 3.x application calls an entry point in the 16-bit DLL, the 16-bit DLL uses the 32-bit Universal Thunk callback to pass the data over to the 32-bit side. Once the call has been received on the 32-bit side, the proper Win32

DLL entry point can be called.

Note that the components labeled Win32 DLL (UT) and Win32 DLL in the diagram above can be contained in the same Win32 DLL. Remember that the code in the Win32 DLL (UT) portion isn't supported under Windows NT, so this code must be special-cased if the DLLs are combined.

For more information, please see the Win32s "Programmer's Reference."

Additional reference words: 3.10 reverse universal thunk

INF: Win32s Message Queue Checking

Article ID: Q97918

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Win32 applications that are designed to run with Win32s need to check the message queue through a GetMessage() or PeekMessage() call to avoid locking up the system. With Windows NT this is not a problem because the input model is desynchronized. That is, each thread has its own input event queue rather than having one queue for the entire system. In a synchronous input model, one application can block all of the others by allowing the single system queue to fill up with its messages. With Windows NT, an application that lets its input queue fill up will not affect other applications.

Additional reference words: 3.10

PRB: _getdcwd() Returns Incorrect Information Under Win32s
Article ID: Q98286

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SYMPTOMS

=====

The following code segment using the C run-time library routine _getdcwd() always returns the root:

```
_getdcwd (3, cBuf, MAX_PATH);  
MessageBox (hWnd, cBuf, "Drive 3 <C drive>", MB_OK);
```

CAUSE

=====

The C run-time library routine _getdcwd() returns incorrect information for the current directory under Win32s.

RESOLUTION

=====

The following code fragments work fine under Win32s:

```
_getdcwd (0, cBuf, MAX_PATH);  
MessageBox (hWnd, cBuf, "Drive 0 <default drive>", MB_OK);
```

-or-

```
GetCurrentDirectory (sizeof (cBuf), cBuf);  
MessageBox (hWnd, cBuf, "SCD", MB_OK);
```

MORE INFORMATION

=====

Running the following sample to display the current directory of drives C and D under Windows NT properly displays the full path of the drive. Running the sample under Win32s always displays the root ("C:\", "D:\").

Sample Code

```
#include <direct.h>  
...  
  
status = _getdcwd(3, szPath, MAX_PATH); // drive 3 == C:  
if (status != NULL) {  
    MessageBox( hWnd, szPath, "Current working directory for C:",
```

```
        MB_OK);
    }

    status = _getdcwd(4, szPath, MAX_PATH); // drive 4 == D:
    if (status != NULL) {
        MessageBox( hWnd, szPath, "Current working directory for D:",
            MB_OK);
    }
    ..

```

Additional reference words: 3.10

KBCategory:

KBSubcategory:

PRB: GetVersion() Returns Invalid Value Under Win32s
Article ID: Q98723

The information in this article applies to:

- Beta Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

In Win32s version 1.0, GetVersion() returns a value of 0x80000A3f, which equates to 3F.0A--Win32s version 61.10.

CAUSE

An internal build number was OR'ed into the low word rather than the high word of the value returned for the version.

RESOLUTION

This bug only affects applications that ship with Win32s version 1.0. Applications that plan to ship Win32s version 1.0 applications can check the return value and translate it to the correct value.

Additional reference words: 3.1 3.10 1.00 1.0

INF: Debugging Win32s Applications

Article ID: Q102430

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

To start debugging a Win32s application, make sure that it runs correctly under Windows NT. Use either WinDbg or NTSD to track down any problems.

Then, install the debugging libraries for Windows 3.1 and the debug version of Win32s. Hook the machine to another machine running a terminal emulator and watch for any warnings that are issued. Be sure to select the Win32sDebug flags carefully--selecting too many will generate more information than you may care to see; selecting too few may cause you to miss important information and warnings.

If you need to debug on Win32s, there are currently two options:

- Use wdeb386 (note that this method is very tricky).
- Use remote WinDbg (WinDbgRm) if you are not familiar with using a kernel debugger. This method requires two machines: a Win32s machine to run the application and WinDbgRm and a Windows NT machine to run WinDbg.

If you have Microsoft Visual C++ for Windows NT, CodeView for Win32s is an additional option. CodeView for Win32s is a user-level debugger; remote debugging is not necessary, and therefore CodeView for Win32s does not require a second machine.

MORE INFORMATION

=====

When performing remote debugging, make sure that the cable is set up exactly as specified in the Win32s Programmer's Reference. The WinDbgRm from Win32s 1.0 does not support software flow control, so it is very important that the hardware flow control is set up properly. If it is not set up correctly, you will have problems as the buffers overflow.

In later versions of WinDbg, XON/XOFF (software) flow control is supported, which means that the standard 3-wire cable can now be used, although the default is still hardware handshaking (5-wire cable). To enable XON/XOFF, you must specify the XON flag in the serial transport parameters on both WinDbg and WinDbgRm.

To enable XON/XOFF on WinDbgRm:

1. Select Options to bring up the Transport dynamic-link library (DLL) dialog box.
2. Select the serial transport and make any needed modifications to the communications port or baud rate parameters.
3. Place the XON flag at the end of the Parameters box. For example, "COM1:19200 XON". Note that the space is needed.

To enable XON/XOFF on WinDbg:

1. Select Options/Debug DLLs.
2. Select the proper serial transport layer.
3. Choose the Change button.
4. Add XON to the end of the Parameters line: "COM1:19200 XON".

It is very important that both sides of the debugger use the same protocol. If they do not, both debuggers will probably hang.

Additional reference words: 3.10

INF: GetCommandLine() Under Win32s

Article ID: Q102762

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

Under Win32s, GetCommandLine() includes the full drive/path of the executable, while under Windows NT GetCommandLine() does not include the full path.

When programs are run from the Program Manager or the File Manager on Windows 3.1, they are spawned using the full path. As a result, argv[0] will have the complete path. When a Win32s application is spawned by a 16-bit application, Windows detects that the application is a Win32s application. The full path is passed to Win32s regardless of whether or not WinExec() was invoked with the full path. As a result, 32-bit applications receive the full path.

When a 32-bit application is spawned from a 32-bit application, the 32-bit kernel passes the information as given by the parent process [that is, if a 32-bit program is started via CreateProcess() from another Win32 application, argv[0] will contain the path that the spawning application passed in].

Additional reference words: 3.10

INF: Win32s Cannot Support _environ in DLLs

Article ID: Q105680

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The `_environ` variable is a pointer to the strings in the process environment, and is used by `getenv()` and `putenv()`. The `_environ` variable cannot be supported in Win32s dynamic-link libraries (DLLs) because per-process data is not available. It is not possible to represent the unique environment of each application in this situation.

Attempting to use `_environ` in a DLL under Win32s in an application that was linked with CRTDLL results in the following error:

```
The procedure entry point "_environ_dll" could
not be located in the Dynamic Link Library
"CRTDLL.dll".
```

Instead, use `GetEnvironmentStrings()` and `GetEnvironmentVariable()`.

Additional reference words: 3.10

INF: Debugging Universal Thunks

Article ID: Q105756

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

The general recommendation for an application targeted for Win32s is to debug it under Windows NT, then make sure that the application works under Win32s. However, Universal Thunks are not supported on Windows NT, so it is not possible to debug applications that use the Universal Thunk.

To debug across the Universal Thunk, you can use WDEB386, which is available with the Windows 3.1 Software Development Kit (SDK). If you are not familiar with WDEB386, you may find it simpler to use other methods. In that case, be sure to install the debug version of Windows 3.1 and the debug version of Win32s and enable suitable notifications for Win32s (unimplemented functions and messages, verbose, and so forth). You may find `OutputDebugString()` useful for displaying extra information.

For more information on WDEB386, please see the Knowledge Base article "Tips On Installing WDEB386." For information on installing the debug version of Windows, please see your Windows SDK documentation.

Additional reference words: 3.10

INF: Using Windows Sockets Under Win32s and WOW

Article ID: Q105757

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Win32s versions 1.1 and later provide a thunking layer for Windows Sockets. A 16-bit Windows Sockets 1.1 package must be installed on the Windows machine. Otherwise, the system will report that WINSOCK.DLL was not found. Windows NT provides a versions 1.0- and 1.1-compliant WOW (Windows on Win32) thunking layer.

There are a number of vendors that sell Windows Sockets packages. Windows Sockets support is available from Microsoft for LAN Manager version 2.2 for MS-DOS and Windows 3.1 at no additional cost. Similar support is also being shipped in "Microsoft TCP/IP for Windows For Workgroups" and the "Microsoft Network Client."

There are two Internet mailing lists for Windows Socket programming information. The following is information on how to subscribe as of this date (10/93):

Send mail to listserv@sunsite.unc.edu with a body that has SUBSCRIBE WINSOCK <your_full_internet_email_address>. If you want to be on the winsock hackers mailing list, substitute WINSOCK_HACKERS for WINSOCK in a separate piece of email.

Additional reference words: 3.10

INF: Win32s and Windows NT Timer Differences

Article ID: Q105758

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Under Windows NT, timers are system objects; as such, they are not owned by an application. SetTimer() can be called from within one application with a handle to a window that was created by a different application. This application would process the WM_TIMER messages in the window procedure. The timer event will continue to occur even after the application that created the timer has terminated. Note that it is fairly uncommon for a Win32 application to create a timer for another application, but this method does work.

Because Win32s runs on top of Windows 3.1 and shares many of its characteristics, timers are owned by the application that calls SetTimer(). The timer event terminates when the application that owns the timer terminates.

Additional reference words: 3.10

INF: Using Serial Communications Under Win32s

Article ID: Q105759

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Windows NT and Windows provide significantly different serial communications application programming interfaces (APIs). Win32s does not support the Win32 Communications API.

A good approach to take in writing an application for Win32s that uses serial communications is to create a pair of dynamic-link libraries (DLLs) with the same name. One DLL will use Win32 Communications APIs and be installed under Windows NT. The other DLL will use the Universal Thunk to call a 16-bit DLL that will call the Windows Communications API. This DLL will be installed under Win32s.

For more information on the Universal Thunk, see the "Win32s Programmers Guide" included with the Software Development Kit (SDK). In addition, there is a sample in MSTOOLS\WIN32S\UT\SAMPLES\UTSAMPLE.

Additional reference words: 3.10 comm

INF: Using VxDs and Software Interrupts Under Win32s
Article ID: Q105760

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Calling VxDs directly from Win32 code is not supported under Win32s. Win32s does not support the VxD interfaces, so the call is handled by the underlying Windows system. The Win32 application runs with 32-bit stack and code sections, but Windows expects only 16-bit segments. Therefore, the calls to the VxD cannot be handled by Windows as expected.

To call software interrupts (such as Interrupt 2F) from a Win32 application running under Windows 3.1 via Win32s, place the call in a 16-bit dynamic-link library (DLL) and use the Universal Thunks to access this DLL. To convert the addresses between segmented and linear address, use `UTSelectorOffsetToLinear()` and `UTLinearToSelectorOffset()`.

Additional reference words: 3.10

INF: Getting Resources from 16-Bit DLLs Under Win32s
Article ID: Q105761

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

A Win32 application running under Win32s can load a 16-bit dynamic-link library (DLL) using LoadLibrary() and free it with FreeLibrary(). This behavior is allowed primarily so that GetProcAddress() can be called for printer driver application programming interfaces (APIs).

Calling FindResource() with the handle that LoadLibrary() returns to the DLL that it just loaded results in an access violation. However, the Win32 application can use the following APIs with this handle

- LoadBitmap
- LoadCursor
- LoadIcon

because this results in USER.EXE (16-bit) making calls to KERNEL.EXE.

If you go through a Universal Thunk to get raw resource data from the 16-bit DLL, it is necessary to convert the resource to 32-bit format, because the resource format is different from the 16-bit format. The 32-bit format is described in the Software Development Kit (SDK) file DOC\SDK\FILEFRMT\RESFMT.TXT.

To determine whether a DLL is a 32-bit or 16-bit DLL, check the DLL header. The DWORD at offset 0x3C indicates where to look for the PE signature. Compare the 4 bytes there to 0x00004550 to determine whether this is a Win32 DLL.

Additional reference words: 3.10

INF: Sharing Memory Between 32-Bit and 16-Bit Code on Win32s
Article ID: Q105762

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Memory allocated by a Win32 application using GlobalAlloc() can be shared with a 16-bit Windows application on Win32s. If the memory is allocated with GMEM_MOVEABLE, then GlobalAlloc() returns a handle and not a pointer. The 16-bit application can use the low word of this handle. The high word is all zeros. Make sure to lock the handle using GlobalLock() in the 16-bit application to get a pointer.

NOTE: GlobalAlloc (GMEM_FIXED...) is not the same as GlobalFix [GlobalAlloc (GMEM_MOVEABLE...)]. GMEM_FIXED will allocate locked pages, which is most often not what you want.

Memory allocated by a 16-bit application via GlobalAlloc() must be fixed via GlobalFix() and translated before it can be passed to a Win32 application. Whenever a Windows object is passed to a Win32 application by its 32-bit address, the memory must be fixed, because the address is computed from the selector base only once. If Windows moves the memory, the linear address used by the Win32 application will no longer be valid.

If you are using the Universal Thunk, you can also pass a buffer from a Win32 application to a 16-bit dynamic-link library (DLL) in the UTRegister() call. The address is translated for you. Another alternative is the translation list passed to the callable stubs. Addresses passed in the translation list will be translated during the thunking process. For more information on the Universal Thunk, please see the "Win32 Programmer's Reference."

NOTE: The ability to share global memory handles under Win32s is a result of the implementation of Windows 3.1, in which all applications run in the same address space. This is not true of existing Win32 platforms and will not be true of future Win32 platforms.

Additional reference words: 3.10

Sample: Security API Functions Demonstration

Article ID: Q85397

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SIDCLN sample demonstrates some of the Win32 security API functions, and provides a sample of how a utility could be written that recovers on-disk resources remaining allocated to deleted user accounts.

More Information:

The on-disk resources recovered are:

Files that are still owned by accounts that have been deleted are assigned ownership to the account logged on when this sample is run.

ACEs for deleted accounts are edited (deleted) out of the ACLs of files to which the deleted accounts had been granted authorizations (eg., Read access)

It may be that running this sample as a utility has no practical value in many environments, as the number of files belonging to deleted user accounts will often be quite small, and the number of bytes recovered on disk by editing out ACEs for deleted accounts may well not be worth the time it takes to run this sample. The time it takes to run this sample may be quite significant when processing an entire hard disk or partition

Note: This sample is not a supported utility.

TO RUN:

You must log on using an account, such as Administrator, that has the privileges to take file ownership and edit ACLs

The ACL editing part of this sample can only be exercised for files on a partition that has ACLs NT processes: NTFS

Typical test scenario: Create a user account or two, log on as each of these accounts in turn, while logged on for each account, go to an NTFS partition, create a couple of files so the test accounts each own a few files, use the file manager to edit permissions for those files so that each

test user has some authorities (e.g., Read) explicitly granted for those files. Logon as Administrator, authorize each test user to a few Administrator-owned files. Delete the test accounts. Run the sample in the directories where you put the files the test accounts owned or were authorized to.

Sample: Saving/Loading Bitmaps in .DIB Format on MIPS
Article ID: Q85844

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SYMPTOMS

In Win32, saving or loading a bitmap in .DIB file format is basically the same as in Win16. However, care must be taken in DWORD alignment, especially on the MIPS platform.

An exception occurs when loading or saving a bitmap on the MIPS platform. In NTSD, the following error message is received:

```
data mis-alignment
```

CAUSE

A non-DWORD aligned actual parameter was passed to a function such as GetDIBits().

The .DIB file format contains the BITMAPFILEHEADER followed immediately by the BITMAPINFOHEADER. Notice that the BITMAPFILEHEADER is not DWORD aligned. Thus, the structure that follows it, the BITMAPINFOHEADER, is not on a DWORD boundary. If a pointer to this DWORD misaligned structure is passed to the sixth argument of GetDIBits(), an exception will occur.

RESOLUTION

To resolve this problem, copy the data in the structure over to a DWORD-aligned memory and pass the pointer to the latter structure to the function instead. See the sample code LOADBMP.C for detail.

More Information:

There is a sample to illustrate this process. Refer to the LOADBMP.C file in the MANDEL sample.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

Sample: Common Dialog DLL

Article ID: Q81703

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A sample demonstrating the use of all of the common dialog box functions in the Win32 API is now available.

More Information:

Each dialog box is demonstrated being used in three different ways: standard, using a hook function, and using a modified template.

Additional reference words:

ChooseColor, ChooseFont, GetOpenFileName, GetSaveFileName

Sample: Determining Drive and File System Type

Article ID: Q81719

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A demonstration of how the `GetDriveType` and `GetVolumeInformation` functions determine all logical drives on a system, their disk type (local, remote, CD-ROM, and so on), and their file system type (FAT, HPFS, and so on) is available in a sample file named `DRIVES`.

More Information:

The additional API functions, `GetLogicalDrives` and `GetLogicalDriveStrings`, are not required to determine the drive and file system type, but are included as an example of how these API functions can enhance the efficiency of disk querying API calls.

When a drive type is removable (for example, a floppy disk drive), then additional precautions are taken before accessing this drive. A validation check is made to see if media exists in the drive before proceeding. A simple test of opening any file in the root directory of the removable media drive using the `OpenFile` API function determines the media's presence. If the `OpenFile` call returns a handle, then media is present and further disk querying calls are safely made on the logical drive. If the `OpenFile` call fails, then no media is present and no further attempts to query this drive are allowed. Note: in order to eliminate an unwanted pop-up, prompting the user to insert a disk in the drive, from being generated by the operating system, the error mode is temporarily adjusted, using `SetErrorMode`, to allow any `OpenFile` errors to immediately return to the calling routine.

Sample: Walking a Directory Tree

Article ID: Q81720

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

A demonstration of how to recursively find all subdirectories under the current working directory is available in a sample file named WALK.

The WALK sample can be found in the \Q_A\SAMPLES\WALK directory.

More Information:

Starting with the current working directory, a call is made to the Walk function which will find all subdirectories in the current working directory. When a subdirectory is found, the current working directory is changed to this subdirectory and another, recursive call is made to Walk, which again will find all subdirectories in this new current working directory. Once all subdirectories for the current working directory have been found, the current working directory is changed up one level (..). When the original current working directory is re-entered, then the recursive process stops.

Additional reference words:

FindFirstFile, GetCurrentDirectory, SetCurrentDirectory
FindNextFile, GetFileAttributes

Sample: World Coordinate Transform

Article ID: Q81721

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SDK sample named WXFORM provides a demonstration of the new world-coordinate transformation. This sample displays a rectangle in world coordinates and a matrix containing the transform values. Users can directly manipulate the rectangle and see the effect on the transformation, or they can set the transformation and see the effect on the rectangle.

More Information:

The program begins by setting the viewport origin to the center of the client area. It then draws a rectangle in world coordinate space from the point (0, 0) to the point (100, 100). The user can directly manipulate this rectangle by using the left and right mouse buttons. Specific actions are described more fully in the "Direct Manipulation Help" dialog box.

There is a second dialog box titled "World Transform." This shows the values of the eM11, eM12, eM21, eM22, eDx, and eDy fields in the XFORM structure retrieved by calling the GetWorldTransform function. By choosing the buttons on this dialog box, the user can cause a SetWorldTransform to occur in the program.

There are three coordinate systems of interest in this sample. The first one is the world coordinate system, which is new to Win32. These points are ultimately mapped to the second coordinate system, device coordinates, before being painted in the window. This program must also use a third coordinate system, screen coordinates, for certain interactions with the mouse pointer.

There is a third dialog box titled "Mouse Position" that shows the location of the cursor in all three of these coordinate systems. The device coordinates are relative to the upper-left corner of the client area. They are not relative to the viewport origin.

Additional reference words: `ModifyWorldTransform`

Sample: AngleArc Demonstration Program

Article ID: Q81724

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The sample named ANGLE provides a demonstration of how the new AngleArc API function works. The X, Y, and RADIUS parameters are all in the world coordinate space. The start angle and sweep angle are floating-point values and are interpreted as degrees.

More Information:

This program presents a dialog box stretched across the top of the window. The user can set the parameters for the AngleArc API function by changing the values in the entry fields of this dialog box. A button on the dialog box then allows the user to immediately see the results of these values on the arc in the client area. If the values in the entry field are invalid, the program will write out this information and not draw the arc. The origin of the viewport is shifted down in the client area so that it exists at the upper-left corner of the viewable area.

Sample: Using GetDIBits() for Retrieving Bitmap Information
Article ID: Q85846

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

When saving a bitmap in .DIB file format, the GDI function is used to retrieve the bitmap information. The general use of this function and the techniques for saving a bitmap in .DIB format are largely unchanged; however, this article provides more details on the use of the Win32 API version of the GetDIBits() function. MANDEL is a sample program that illustrates the information in this article.

More Information:

The function can be used to retrieve the following information:

- Data in the BitmapInfoHeader (no color table and no bits)
- Data in the BitmapInfoHeader and the color table (no bits)
- All the data (BitmapInfoHeader, color table, and the bits)

The fifth and the sixth parameters of the function are used to tell the graphics engine exactly what the application wants it to return. If the fifth parameter is NULL, then no bits will be returned. If the biBitCount is 0 (zero) in the sixth parameter, then no color table will be returned. In addition, the biSize field of the BitmapInfoHeader must be set to either the size of BitmapInfoHeader or BitmapCoreHeader for the function to work properly.

Refer to the SAVEBMP.C file in the MANDEL sample for details.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

Sample: Writing NTSD Extensions

Article ID: Q85885

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

This article and the associated sample (called NTSD) demonstrate how to write an NTSD extension for the NTSD debugger.

While debugging, it is often necessary to look up certain fields of a particular structure in the program. This process usually involves dumping the address of the structure in question and locating the specific fields in the dump, which can be tedious and inefficient.

With NTSD, programmers can write a dumping routine to be called by the NTSD debugger.

The routine must be in a DLL and have the following prototype:

```
void Routine (HANDLE, HANDLE, HANDLE, PNTSD_EXTENSION_APIS, LPSTR);
```

See the file DEBUG.C for details.

Then, to invoke the routine in NTSD, the user would do the following:

```
!module.routine argument
```

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

SAMPLE: Process API Functions Example

Article ID: Q81825

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The PROCESS sample application provides a simple interface to the `CreateProcess()` and `TerminateProcess()` functions. To create a process, the user is presented with a common dialog box for selecting a file. In this case, the file must have an extension of `.EXE`. Processes that are started are presented in a list box. Any of the processes can be selected in the list box and then terminated.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at (800) 227-4679, ext 11771.

Warning: "TerminateProcess() is used to cause all of the threads within a process to terminate. While `TerminateProcess()` will cause all threads within a process to terminate, and will cause an application to exit, it does not notify DLLs that the process is attached to that the process is terminating. `TerminateProcess()` is used to unconditionally cause a process to exit. It should only be used in extreme circumstances. The state of global data maintained by DLLs may be compromised if `TerminateProcess()` is used rather than `ExitProcess()`."

Additional reference words: 3.10

Sample: Using Thread API Functions

Article ID: Q81826

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The THREADS sample application shows how to use SetThreadPriority, SuspendThread, and ResumeThread.

More Information:

To use THREAD, start the application. Two threads will be created; one will draw a red rectangle and the other a green rectangle. Both of these rectangles move about the window; their speed and behavior is based on the thread priority and its resumed/suspended status. The priority and status are set through menu selections.

Of special interest is the suspension count. The system keeps track of the number of times a thread has been suspended and resumed. Each time the thread is suspended, the count is incremented; each time it is resumed, the count is decremented. The suspension count can either be tracked by applications manually, the same way this sample application does, or the return value from ResumeThread and SuspendThread can be used to obtain the previous suspension count before the call was made. Only when the suspension count is zero will the thread run.

A thread now has seven levels of priority exposed at the API level:

```
THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL
```

Note, caution should be used when setting a thread priority to THREAD_PRIORITY_TIME_CRITICAL. This level is high enough to interfere with the application's window performance, and interfere with other applications running on the system.

Sample: Demonstration of Using System Info API
Article ID: Q81849

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The GETSYS SDK sample is a dialog box that provides the user with an easy way to see the results of the following API functions:

- GetSysColors()
- GetSystemDirectory()
- GetSystemInfo()
- GetSystemMetrics()
- GetSystemPaletteEntries()
- GetSystemTime()

Sample: StretchBlt Demonstration

Article ID: Q81850

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The STREBLT sample is an easy to use demonstration of the StretchBlt API function. The program presents a dialog box on the top of the window, and through this dialog box the user can manipulate the parameters to StretchBlt. In the main window, the source bitmap is displayed on the right half of the window, and the destination bitmap is displayed on the left half.

More Information:

The source and destination rectangles may be changed directly in the dialog, or they may be changed by using the direct manipulation objects in the two halves of the window. Clicking and dragging the mouse in the upper-left corner moves the rectangles; clicking and dragging the mouse in the lower-right corner sizes the rectangles. The source direct manipulation object is temporarily erased before calling StretchBlt so that the top and left edges do not show in the destination image.

The raster operation for the StretchBlt call may be changed by altering the values in the right-most entry fields. The contents are interpreted to be in hexadecimal. There is a combo box directly beneath these entry fields that lists all of the standard raster operations. If the user selects a standard ROP from this combo box, its contents are copied into the ROP entry fields and are then used in the StretchBlt call.

Several of the raster operations make use of a pattern in the destination HDC. For this reason, the program also allows the user to select one of the standard pattern brushes from a second combo box. This brush is selected into the destination HDC just prior to making the StretchBlt call.

The effect of the StretchBlt call is also affected by the "StretchBlt mode" that has been set for the destination HDC. A third combo box allows the user to select from any of the standard modes. The difference is most easily observed when stretching from a large source rectangle to a small destination rectangle.

The "Draw" button may be chosen at any time to cause the StretchBlt call to be made. This does not erase the background, so that the effect of multiple ROPs on the HDC can be observed. Manipulating the source rectangle also causes a StretchBlt to occur without erasing the window. However, manipulating the destination rectangle erases the destination half of the window before the next StretchBlt is called.

Sample: Creating Resource-Only DLLs

Article ID: Q85915

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The RESDLL sample shows how to create a resource-only dynamic link library (DLL). In short, this is accomplished by creating and resource-compiling a resource (.RC) file, and then linking it correctly.

The MAIN.EXE program tests THE_DLL.DLL by loading it and referencing the DLL's icon, cursor, and bitmap. The icon and cursor are used by the registered window class, and the bitmap is used in painting the client area.

SAMPLE: Standard DLL & Ex. of Creating a Custom Control Class
Article ID: Q81852

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

The SPINCUBE sample provides a generic Windows NT dynamic link library (DLL) template demonstrating the use of DLL entry points, exported variables, using C run time in a DLL, and so forth.

This sample also provides a functional example of how to create a custom control class that may be used by applications (for exampl, SPINTEST.EXE) as well the Dialog Editor.

MORE INFORMATION

=====

SPINCUBE.DLL contains the control window procedure and the interface functions required by the Dialog Editor (see SPINCUBE.C), as well as the control paint routines (see PAINT.C). SPINTEST.EXE is a small test program that loads SPINCUBE.DLL and creates a few of the custom controls.

To test SPINCUBE with the Dialog Editor:

1. Start the editor. From the File menu, choose Open Custom.
2. Enter the path and filename of SPINCUBE.DLL.
3. Create a new dialog box and choose a custom control button from the control palette (lower-right corner).
4. Click the dialog box to create a SPINCUBE control.
5. Save the dialog box template.
6. Inspect the .DLG file that was created.

Additional reference words: 3.10

Sample: Using Region-Related API Functions

Article ID: Q81874

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The REGIONS sample demonstrates various region-related API functions, and allows a user to create rectangular, elliptic, and multi-polygon regions. In addition, hit-testing a region and combining regions using the different region-combination modes is demonstrated.

More Information:

When the program has started, create a region by choosing one of the items in the Create submenu. At this point, items in the Options submenu will be enabled, and hit-testing, inversion, and other actions can be performed on the region.

When a second region is created, items in the Combine submenu will be enabled. Choosing one of these items causes CombineRgn to be called with the specified combine mode, and the two regions are merged into one.

It is possible to create up to three regions at a time. Items in the Options submenu always apply to the most recently created (or combined) region. The Erase item deletes all existing regions. Items in the Combine submenu always apply to the two most recently created (or combined) regions.

Additional reference words:

PtInRegion, CreateEllipticRgn, GetRgnBox, CreatePolygonRgn, CreateRectRgn, SetRectRgn, OffsetRgn, FillRgn, FrameRgn

Sample: PlgBlt Demonstration

Article ID: Q81875

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

Win32 offers a new API function that will copy a bit image onto an arbitrary parallelogram. Now, for the first time, application programs can trivially rotate or shear bitmaps. The PLGBLT SDK sample is an easy to use demonstration of how this new API function may be used.

More Information:

The program presents a dialog box on top of the window that displays the input parameters to the PlgBlt function. By choosing the "New Src" or "New Mask" button, the user can select a new bitmap for use as the source bitmap or as the monochrome mask bitmap. The client area of the window is divided into three regions. The region on the left contains the result of the PlgBlt operation. The region in the middle provides the source HDC, and the region on the right provides the mask bitmap for the PlgBlt operation.

In each of the three regions, there is a "direct manipulation object." This object may be picked up and moved by clicking the left mouse button in the top-left corner and dragging. The three objects are restricted in their response to user actions to correctly reflect the parameters to the PlgBlt function. The object in the mask region may be moved only. The object in the source region may be moved or sized. The object in the destination region may be moved, sized, sheared, or rotated. Please see the WXFFORM sample for more information on how this direct manipulation is accomplished. Additional information on the WXFFORM sample may be obtained by querying on the word WXFFORM in this knowledge base.

SAMPLE: Using Graphic Paths Demonstration

Article ID: Q81876

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The PATHS sample demonstrates the use of paths for drawing, filling, and clipping. The program draws six different figures in the window and labels each one. Each figure is based on the same path re-created six times. The six figures are the result of calling the following Windows functions (with the poly fill mode in parentheses):

```
StrokePath()  
FillPath()  
StrokeAndFill() (Winding)  
StrokeAndFill() (Alternate)  
SelectClipPath() (Winding)  
SelectClipPath() (Alternate)
```

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

Additional reference words: 3.10

SAMPLE: PolyBezier() Demonstration

Article ID: Q81877

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The BEZIER sample provides an easy to use demonstration of how the PolyBezier() function works. The user can place points in the window with the left mouse button. The user can also move these points by dragging with the same mouse button. The PolyBezier() curve is drawn dynamically to follow the position of the new points.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

In order to use this program, press the left mouse button at miscellaneous places in the client area. A Polyline() call shows exactly where the points were put. When there are 4, 7, 10, ..., (3n+1) points on the screen, the PolyBezier() curve is drawn with these as control points. The API itself does not draw anything if there are some other number of points. The whole client area may be erased by pressing the right mouse button.

Additional reference words: 3.10

Sample: Demonstration of Setting File Attributes
Article ID: Q85917

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SETINFO sample shows how to get and set information on file date, time, attributes, and size. SETINFO demonstrates much of the functionality of OS/2's DosQFileInfo() and DosSetFileInfo().

More specifically, the SETINFO sample shows how to set file attributes and how to modify file and date times (and how to do the conversions from file time to DosTime, and so on). To use the sample file, enter a filename in the appropriate edit field and choose the Get Info. button. To set file attributes or file date and time information, modify the values in the edit fields and check buttons, and choose the Set Info. and Set Attr. buttons. To slow down the return code reporting, enter a larger value into the time edit field, and choose the Set Time button.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

Sample: GetDeviceCaps() Demonstration Program
Article ID: Q83930

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The GETDEV sample is a dialog box that displays the result of the GetDeviceCaps call with all of the possible input parameters. Six of the numeric results (TECHNOLOGY, LINECAPS, POLYGONALCAPS, TEXTCAPS, CLIPCAPS, and RASTERCAPS) are expanded to show the constant string from WINGDI.H.

Sample: PolyDraw Function Demonstration

Article ID: Q83931

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The POLYDRAW sample provides an easy-to-use demonstration of how the PolyDraw Win32 API function works. The user can place points in the window with the left mouse button, and move these points by dragging with the same mouse button. The PolyDraw curve is drawn dynamically to follow the position of the new points.

More Information:

To use this program, click the left mouse button at miscellaneous places in the client area. A Polyline call shows exactly where the points were put. By default, the type entered into the type array is PT_LINETO. This can be changed to a PT_MOVETO type by holding down the SHIFT key. It can be changed to a PT_BEZIERTO type by holding down the CTRL key. The resulting purple curve shows the results. There will be no curve when the bezier points do not come in groups of 4, 7, ... , (3n+1).

SAMPLE: Simple DLL Demonstration

Article ID: Q83932

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SIMPLDLL sample provides a generic DLL template. Also included are two small test applications, LINKTEST and LOADTEST, which demonstrate load-time linking (to a DLL import library) and dynamic loading, respectively.

More Information:

THE_DLL contains a skeleton DLL (dynamic-linked library) entry point and five exported functions with varying parameter lists. A resource file (containing a dialog box template) is also used.

LINKTEST is a small application that links with the THE_DLL's import library, and allows the user to make calls into THE_DLL (via menu item selections).

LOADTEST is a small application that loads THE_DLL at run time and calls the GetProcAddress function to retrieve the addresses of THE_DLL's exported functions. Again, the user is allowed to make calls into THE_DLL.

Additional reference words: GetModuleFileName, LoadLibrary, GetProcAddress

Sample: Using Timers in Windows NT
Article ID: Q83933

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The TIMERS sample illustrates the use of the SetTimer and KillTimer Win32 API functions. Button controls allow a user to start and stop timers, which determine the frequency of flashing rectangles in the client area.

Sample: Using Anonymous Pipes to Capture Child Process Output
Article ID: Q84082

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The sample INHERIT demonstrates how to redirect standard output and standard error to an anonymous pipe using two different techniques: passing anonymous pipe handles to the child in the STARTUPINFO structure, and by setting anonymous pipe handles as the standard handles with the SetStdHandle API and having the child process inherit them.

In the sample, a child process is started whose standard output and standard error handles have been redirected to an anonymous pipe. The parent reads out of this pipe and puts the data to both the console and to a log file specified on the command line.

Additional reference words:

CloseHandle CreateProcess ReadFile
CreateFile GetLastError CreatePipe
WriteFile

Sample: Virtual Memory API Function Demonstration
Article ID: Q85919

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

VIRTMEM is a sample of the various virtual memory API functions available under Win32.

When you start the application, you are initially given a RESERVED page of virtual memory. You can change the protection and state of the page through menu selections. Check marks will appear in the menu items to indicate the current state and protection on the page. More in-depth information regarding the page can be obtained by selecting the Show Page menu item.

The Lock menu item allows you to lock and unlock the page in memory.

The application also uses structured exception handling and allows you to try and write to the page in its various states and protections. To do this, select the Test menu option.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

Sample: SUBCLASS Program Demonstration

Article ID: Q84242

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SUBCLASS sample demonstrates how a program can subclass standard controls in order to extend their normal functionality. This sample replaces the window procedure for buttons, edit fields, and list boxes.

More Information:

The standard subclassing technique is to replace the window procedure in the window structure by using:

```
SetWindowLong (hwnd, GWL_WNDPROC, (LONG) SubclassWndProc);
```

In the SUBCLASS sample, the old window procedure is also saved in a structure pointed at by the user data. Thus, any functionality can be added to various classes of windows without having to know what the class originally was.

In this sample, the subclass procedure adds the ability to move and size the control windows when the application is not in "test mode." When the application is in test mode, the subclass procedure calls the original window procedure and the controls behave as normal. Thus, this sample provides the bare bones for a "dialog editor" type of program.

Additional reference words: CallWindowProc GWL_USERDATA

Sample: CreateProcess() Priority Demonstration

Article ID: Q84539

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The sample STARTP demonstrates how to start a new process at a given default priority. It is a functional replacement for the "start" command, but with added features.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

STARTP starts a separate window to run a specified program. The following is the command syntax for STARTP.

```
STARTP [/Ttitle] [/Dpath] [/h] [/l] [/min] [/max]
        [/c] [program] [parameters]
```

title	Title to display in window title bar. Put entire parameter in quotation marks to include spaces in the title; for example, startp "/Ttest job".
path	Starting directory.
h	Set default to high priority.
l	Set default to low priority.
min	Start window minimized.
max	Start window maximized.
c	Use current console instead of creating a new console.
program	A program to run as either a GUI application or a console application.
parameters	These are the parameters passed to the program.

Note that the priority parameters may have no effect if the program changes its own priority.

Sample: Code Demonstration to Put a DACL on Floppy Disk Drives
Article ID: Q99459

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

There is no procedure for putting a discretionary access control list (DACL) on floppy disk drives or COM ports with REGEDT32, using the Control Panel, or using another part of the user interface. Furthermore, there is no procedure for using the Win32 API to put a DACL that survives reboots onto floppy disk drives or COM ports.

However, SD_FLPPY.EXE puts DACLs that survive logoff/logon on floppy disk drives or COM ports; that is, these DACLs are on the floppy disk drives or the COM ports until the next reboot.

A version of this sample program can be installed as a service such that each time the machine starts, the DACLs are automatically re-applied.

Purpose of This Sample

The purpose of this sample is to show sample code that applies DACLs to floppy disk drives or COM ports.

Note: There may be as many desired user interfaces to this sort of functionality as there are people thinking about this, so it is not a purpose of this sample (or the Win32 service variation of it) to present an incredibly cool user interface to how the DACLs get applied. A very simplistic approach is taken to the user interface. Anyone who wants more complicated DACLs applied, or wants other variations in the user interface, will probably benefit by being able to use this sample code as a starting point for their DACL-applying application.

To Run

Type sd_flppy to lock the \\.\A: and \\.\B: devices.

Putting SD_FLPPY.EXE in a startup group or logon script might work for some people.

This is not a supported utility.

More Information:

The version of this program that is packaged as a Win32 service is in this same directory and is built with SD_FLPPY.EXE by the same

makefile.

The packaged-as-a-service approach might better suit people who need to change the DACL on the floppy disk drives without requiring a reboot or logoff. After installing the FLPSDSRV.EXE service on the machine, the client application CHGFLPSD.EXE can be used over the network to lock, unlock, or query the locked state of the floppy disk drives of any machine where the FLPSDSRV.EXE service is running.

This packaged-as-a-service approach might better suit people that want to inquire over the net what DACLs are on the floppy disk drives of particular machines (to check or audit them). And this approach might better suit people who would prefer that the DACLs be applied as the system boots up so that the DACLs are applied before any user has logged on at the keyboard.

The packaged-as-a-service approach is useful for protecting the floppy disk drives as a resource on a particular machine (regardless of who, if anyone, is logged on); whereas, the SD_FLPPY.EXE approach (running an .EXE at logon time) is useful for keeping a particular user from using the floppy disk drives on any machine that that user might use. However, once user Sam6 has logged on to machine \\Mach3 and locked the floppy disk drives with SD_FLPPY.EXE, the floppy disk drives will stay locked until reboot. Of course a utility could easily be written that could run in the startup group of a different authorized user such as Jane3 to force the floppy disk drives on any machine Jane3 logs on to to be unlocked.

Additional reference words: 3.10

Sample: MaskBlt Function Demonstration

Article ID: Q84541

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MASKBLT sample is an easy-to-use demonstration of the MaskBlt Win32 API function. The program presents a dialog box on the top of the window; through this dialog box the user can manipulate the parameters to MaskBlt. In the main window, the source bitmap is displayed in the center third of the window, the monochrome bitmap mask is displayed in the right third of the window, and the destination bitmap is displayed on the left.

More Information:

The destination rectangle may be changed directly in the dialog box, or it may be changed by using the direct manipulation object in the left third of the window. Clicking and dragging the mouse in the upper-left corner moves the rectangle; clicking and dragging the mouse in the lower-right corner sizes the rectangle. The function requires only a starting point (not a rectangle) for the source and mask bitmaps. There is one additional direct manipulation object for the source and one for the mask. These objects may be moved by clicking and dragging with the mouse.

The raster operation for the MaskBlt call may be changed by altering the values in the right most entry fields. The contents are interpreted to be in hexadecimal. There is a combo box directly beneath these entry fields that lists all of the standard raster operations. If the user selects a standard ROP from this combo box, its contents are copied into the ROP entry fields and are then used in the MaskBlt call.

This sample provides clipboard support in the following manner. Hitting <ctrl><insert> will copy the destination image into the clipboard. Hitting <shift><insert> will copy a bitmap from the clipboard into the source region. Hitting <alt><insert> will do both; the destination image will be copied into the clipboard and then down to the source region.

Sample: WNet API Function Demonstration

Article ID: Q87328

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The WINNET sample finds a connectable disk resource on the network, connects to it, then disconnects. It's purpose is to demonstrate the WNet Win32 APIs.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

Requirements for WINNET:

- Ensure one or more network disk shares can be used by the machine/user-ID you run WINNET on.
- Ensure that these one or more network disk shares are not already connected to the machine/user-ID you run WINNET on.
- These net shares must not require a password.

IMPORTANT: Ensure net drive W: is not in use when you run WINNET.

Additional reference words: user ID WNetAddConnection
WNetAddConnection2 WNetCancelConnection WNetCancelConnection2
WNetCloseEnum WNetEnumResource WNetGetConnection WNetGetLastError
WNetGetUser WNetOpenEnum

Sample: Demonstration of Setting Console Text Color
Article ID: Q87329

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The CONSOLEC sample illustrates the use of the SetConsoleTextAttribute() and GetConsoleScreenBufferInfo() functions to set the console text color attributes.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

This sample also functions as a utility to set the text color of the console window. The command syntax for the utility is as follows:

```
COLOR FOREGROUND [BACKGROUND]
```

FOREGROUND and BACKGROUND are the new text color selections for the current console. If the utility is invoked without any options, the utilities syntax and a table of the possible color choices is displayed. The BACKGROUND selection is optional, and thus just the FOREGROUND text color can be changed.

Possible colors are: black, blue, green, cyan, red, magenta, yellow and white. Each of these can be selected as the FOREGROUND or the BACKGROUND color. Selection of the same color for both the FOREGROUND and the BACKGROUND is not permitted. The color options are not case sensitive, and only the first unique characters are necessary to select the color. For example

```
COLOR BLU W
```

will select blue on white text color attributes.

The text color attribute changes only affect new console output. Thus, text in the console buffer before the utility is invoked retains its original color attributes.

Additional reference words:

Sample: Asynchronous I/O Demonstration

Article ID: Q87330

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The EVENT sample demonstrates performing asynchronous I/O in Win32. In Win32 you can do this in two ways. One way is similar to OS/2 where a thread is spawned that performs the I/O and returns. With Win32, when you create a file, it signals to the system that you want to perform I/O asynchronously. Then, when ReadFile(), for example, is going to take a significant amount of time to complete, an ERROR_IO_PENDING error is generated, signaling you to do other tasks until you NEED the data, at which time you can use the GetOverlappedResults() function, which will finish the I/O.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

Note that this activity has taken place without the need of an additional thread to perform the work. This sample touches only on the capabilities of what one can do with the new overlapped I/O functions. For example, an application that uses pipes to communicate over the network to other clients can create these file handles with the overlapped flag. Then, instead of blocking and waiting for a connection, the server application can go about doing useful tasks waiting for the pipe to enter the "signaled" state. In addition, you can perform more than one operation on this handle at one time, such as reading and writing to the same file.

All this power does not come without some responsibilities on the programmer's side. First, the system does not keep track of the system file pointers. In addition, you cannot use the data until the system responds by setting an event to a signaled state.

In the first case this just means you need to keep track of the value "lpNumberOfBytesTransferred" returned by GetOverlappedResult() and update the OVERLAPPED structure with this information. This OVERLAPPED structure will then be passed into the Read/WriteFile() function, which will use this as the offset to the starting point for the I/O operation. The first call to Read/WriteFile() will normally then have the offset fields in the OVERLAPPED structure set to zero.

The second case should be used as a criteria of whether to use this type of I/O. If you need the data before you can do anything else, use

normal synchronous I/O and let the system handle the details for you. This also demonstrates an important reason for using an EVENT to wait on rather than the file handle. While both are allowed in a multithread application, one cannot guarantee that the thread that set the handle to the signaled state will be the one returning from the GetOverlappedResult() because each thread is using the same handle to wait on.

To keep this sample focused, the user interface is simple. To run this sample at the command prompt, type:

```
ASYNC_IO <In_file> <Out_file>
```

In_file and Out_file are place holders. As this is implemented, you cannot write over an existing file. While this is up and running, you will see vital statistics such as, such as the following:

- When I/O is pending
- How many bytes are transferred
- End of file

Sample: Communications API Function Demonstration

Article ID: Q87331

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The COMM SDK sample application is designed to demonstrate the basics of using the Win32 communications API functions while maintaining a common code base with Win16 code.

This sample is included with the Microsoft Win32 Software Development Kit for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at 1 (800) 227-4679, ext 11771.

More Information:

The COMM program performs communications using the Windows functions `OpenFile()`, `ReadFile()`, `SetCommState()`, `SetCommMask()`, `WaitCommEvent()`, `WriteFile()`, and `CloseFile()`.

This sample creates a background thread to watch for COMM receiver events and posts a notification message to the main terminal window. Foreground character processing is written to the communications port.

Simple TTY character translation is performed and a screen buffer is implemented for use as the I/O window.

Overlapped file I/O techniques are demonstrated.

How to Use

The baud rate, data bits, stop bits, parity, port, RTS/CTS handshaking, DTR/DSR handshaking, and XON/XOFF handshaking can be changed under the Settings menu item.

Once the communications settings are set up, the Action menu item can be selected to connect or disconnect the TTY program.

Sample: Demonstration of the Console API Functions

Article ID: Q87332

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The CONSOLE sample demonstrates the Win32 console API functions. The program takes no parameters; to start, just run CONSOLE.EXE. After starting the sample, you can click on one of the functions on the screen to get a demonstration of that function. When viewing a demo of the function, the title of the console window is changed to show the name of the source file where that demo function resides. This should make it easy to find the sample code where the function of interest resides.

Please note that some of the demos cover multiple APIs, so some of the menu choices run the same demo.

Sample: Distributed Bounded Buffer Solution (DBBS)

Article ID: Q95528

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The BNDBUF sample demonstrates the distributed version of the classical operating systems producer-consumer problem. A centralized buffer pool managed by the RPC server is used by the producers and consumers. Counting semaphores are used to make sure that consumptions take place when there is at least one unconsumed string in the buffer pool and productions take place when there is at least one empty slot in the buffer pool. Synchronization to the shared buffer pool is coordinated by means of a mutex.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at (800) 227-4679, ext 11771.

More Information:

To use this program, the RPC locator service must be started first using the following command line:

```
start locator /noservice
```

Next, start the application server by typing:

```
start bndbufs
```

Multiple clients could then be started by typing the following for each client to be started:

```
start bndbufc
```

The application uses while loops to run forever. Therefore, you need to use CTRL+C to terminate each component.

Additional reference words: 3.10

Sample: Demonstrating GDI and User APIs in Fractals

Article ID: Q94898

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MANDEL sample demonstrates the Win32 GDI and USER API functions in the setting of fractals.

More Information:

The Mandelbrot Dream provides the following functions:

- Drawing the Mandelbrot set and the corresponding julia set
- Zooming into any of the set
- MDI fractal drawing windows
- Floating Point Math/Fix Point Math
- Shifting color table entries
- Changing palette entries and animating palatte aka color cycling
- Loading/Saving bitmap created with special effect
- Changing bitmap color with flood fill
- Boundary tracing and creating a clip region out of it for creating special effect
- Enumerate printers for printing
- Load RLE (or convert .bmp files to RLE) for playing in viewer
- Save the RLE in memory to disk.

Note: The sample also makes use of the LARGE_INTEGER library shipped with the SDK for its fixed point math.

Drawing the Mandelbrot Set and the Corresponding Julia Set

- 1.To draw the Mandelbrot set, choose the Mandelbrot Set menu item from the Create menu to create a MDI child window to draw the set in if one has not already been created.
- 2.Then, choose either "use Fix Point math" or "use Floating Point math" from the Draw menu to decide if floating point or fixed point math is desired for calculation. Fix Point is faster, however you lose resolution sooner as you zoom in.
- 3.Also, choose the number of iterations from the Iteration menu item and choose Step from the Draw menu. The higher the number of iterations, the more detail is the picture

but the slower to generate the picture. The step determines whether every scan line is drawn. The more scan lines it has, the better the picture but the slower to generate the picture.

- 4.To start drawing, choose Draw Set from the Draw menu.
- 5.To draw the Julia set, use the right mouse button to select a point in the Mandelbrot set (the drawing surface of the Mandelbrot window). A new Julia MDI window will be created. Then choose Draw Set from the Draw menu to start drawing.
- 6.The point selected with the right mouse button determines the complex constant to use for the Julia Set.

Zooming Into Any of the Set

- 1.To zoom in, click, drag and release with the left mouse button to describe the zoom in region.

A new MDI child of the same type as the parent (Mandelbrot window or Julia window) will be created.

- 2.Choose Draw Set from the Draw menu to start drawing.

MDI Fractal Drawing Windows

Choose either Mandelbrot Set or Julia Set from the Create menu for creating a new MDI window for drawing.

Or, use the left mouse button to describe a zoom in region in either a mandelbrot or Julia MDI window for creating a new MDI window for drawing.

Or, click on the Mandelbrot window with the right mouse button for creating a Julia MDI window corresponding to the mouse click position in the Mandelbrot window.

Floating Point Math/Fix Point Math

Choose the appropriate menu item ("Floating Point math" or "Fix Point math") from the Draw menu. The Fix Point math uses 20.11 fixed point integer arithmetic for calculation.

Shifting Color Table Entries

Choose Shift from the Color menu or hit F10 to shift the color table entry. The picture of the active MDI window will be updated.

Changing Palette Entries and Animating Palette (aka Color Cycling)

Choose Cycle from the Color menu or hit F11 to start color cycling the picture.

The menu item will be grayed if the display device does not support palette management. Currently, only the MIPS display driver supports that.

Loading/Saving Bitmaps Created With Special Effect

Choose Load Bitmap from the Bitmap menu to load a bitmap into the active child window. Or, choose Save Bitmap As to save the picture in the active MDI child window.

Changing Bitmap Color With Floodfill

Choose Custom from the Color menu to select a color. Then the cursor will be changed to a paint can over the active child window. Click with the left mouse button on the picture, the old color under the cursor will be changed to the new color.

Boundary Tracing and Creating a Clip Region Out of it For Creating Special Effect

From any active Mandelbrot window, choose Set Mandelbrot Clip region from the Region menu. The boundary of the escape region will be traced. The region will then be selected as a clip region.

Thus, if you load a bitmap for display, the bitmap will only show through the clip region. The new picture can then be saved.

To remove the clip region, choose Remove Clip Region from the Region menu.

Note, the boundary tracing algorithm may trace out a small island of only several pixels and stop. If that happens, you might change the size of the window or create another zoom window and trace again.

Enumerate printers for printing

On start up, the Mandelbrot Dream will enumerate the printers and insert the printers into the Print menu. Then selecting the printer on the Print menu will print the picture in the active MDI window.

Load RLE (or convert .bmp files to RLE) for playing in viewer

Choose the Viewer item from the Create menu to create a viewer window or bring any existing viewer window to the top. Select Load Bitmap(s) from the File menu for loading RLE or bmp files into the memory from disk. Select the Play or Play Continuously item from the Options menu for viewing.

For demonstration of what this functionality can do. Load the .\rsc\julia.rle file and select Play Continuously.

The Julia.rle is a collections of the various julia sets along the boundary of escaping and non-escaping points of the Mandelbrot Set.

Save the RLE in memory to disk.

Choose the Viewer item from the Create menu to create a viewer window or bring any existing viewer window to the top. Select Save Bitmap(s) from the File menu for saving the RLE(s) from memory to disk.

SAMPLE: Demonstration of the Win32 Font API Functions

Article ID: Q94903

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The TTFONTS sample is an easy to use, powerful utility which allows the user to explore the font system. There is a toolbar on top of the main frame window with buttons that allow the following actions:

- 1.Enumerate all of the fonts installed for the display.
- 2.Get TEXTMETRIC & OUTLINETEXTMETRIC information.
- 3.Create a font based on an arbitrary LOGFONT structure.
- 4.Get "font data" by using the GetFontData() API.
- 5.Enumerate all of the fonts available to the default printer.

More Information:

The program is designed to provide the user an easy interface to the API calls related to the font system. It will not protect against meaningless values, nor will it hide system oddities. Most of the buttons on the toolbar are self explanatory and represent a single system API.

Pressing the EnumFonts button will show all of the face names listed horizontally, and each of the fonts within that face name listed vertically below it. TrueType fonts will be marked with a small colored "TT" bitmap. Fonts that have the DEVICE_FONTTYPE bit on will be marked with a small bitmap image of a printer. When the enumeration windows are showing the user can click the left mouse button to copy the information about a selected font into the LOGFONT and TEXTMETRIC dialogs. The user can dismiss this window without changing the dialog boxes by clicking with the right mouse button or typing any character.

The "Display" window is able to operate in any one of three modes. These are listed in the "Display" menu. The first just writes "Hello" in the middle of the screen, and it grids the background. This is useful when utilizing the lfEscapement and lfOrientation fields of the LOGFONT structure. The second mode writes all of the glyphs between the tmFirstChar and tmLastChar values stored in the TEXTMETRIC structure. The final mode is used only for true type fonts. It calls GetFontData, finds the 'cmap' table, and displays glyphs from the different ranges in this table. Use the horizontal scroll bar in the display window to step through the ranges.

Notice:

There is a font distributed on the Win32 SDK disk which covers more than one thousand unicode characters. In order to use this application to its full potential, you will want to install that font using the Control Panel, Fonts item. The font is named L_10646.TTF.

SAMPLE: A Simple Service

Article ID: Q99460

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The Simple Service sample demonstrates how to create and install a service.

In this particular case, the service merely opens a named pipe, anything, it surrounds the input with

```
Hello! [<input goes here>]
```

and sends it back down the pipe to the client.

The service can be started, stopped, paused, and continued.

To install the service, first compile everything, and then use INSTSRV to install SimpleService as follows:

```
instsrv SimpleService <location of SIMPLE.EXE>
```

To start the service, use either the "net start" method or use the Control Panel Services application.

Once the service has been started, you can use the CLIENT program to verify that the service is working correctly by using the following syntax:

```
client \\.\pipe\simple stuff
```

This should return the response:

```
Hello! [stuff]
```

After using the sample, if you want to remove the service, just type:

```
instsrv SimpleService remove
```

Note: INSTSRV can be cause problems; it can install and remove any service you tell it to, so be careful.

Additional reference words: 3.10 CloseServiceHandle
InitializeSecurityDescriptor SetSecurityDescriptorDacl
SetServiceStatus OpenSCManager StartServiceCtrlDispatcher
RegisterEventSource DeregisterEventSource RegisterServiceCtrlHandler

SAMPLE: Enhanced Metafile Editor

Article ID: Q94904

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MFEDIT sample demonstrates the Win32 enhanced metafile application programming interface (API) functions. This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. For additional information on obtaining a copy of the Win32 SDK, contact the Microsoft Developer Services Team at (800) 227-4679, extension 11771.

More Information:

The MFEDIT metafile editor provides the following functionalities:

- Playing back and recording graphical device interface (GDI) calls.
- Embedding a bitmap and enhanced metafile into another enhanced metafile with transformation.
- Hit-testing against enhanced metafile records.
- Random access playback.
- Playing back metafile records one-by-one.
- Selective recording of existing enhanced metafile records into a new enhanced metafile.
- Drawing with pen, text, bezier, line, ellipse, rectangle, and embedding bitmap and enhanced metafile tools.

Playing Back and Recording GDI Calls

- To play back an existing enhanced metafile, from the File menu, choose Load Metafile, or click the Eject button to bring up the open file dialog box, and select the appropriate file. Then click the Play button to play the file on the drawing surface.
- To record, click the Record button and draw on the drawing surface with the graphic tools provided. When done, click the Stop button.

The GDI calls are recorded as C:\METAFX.EMF, where x is 0, 1, 2, 3, and so forth. If you want to save the metafile with a different name, choose Record Metafile As from the File menu.

The new filename is used as the root for all metafiles recorded; 0, 1, 2, and so forth are appended to the root name.

- The default drawing tool is "pen". To select a different drawing tool, click the desired tool button in the control panel.

Embedding a Bitmap and Enhanced Metafile into Another Enhanced Metafile

with Transformation

- Click the Record button. Select the bitmap or metafile tool and then embed the currently loaded bitmap or metafile as described in 7 below. When done, click the Stop button.

Hit-Testing Against Enhanced Metafile Records

- Play back an enhanced metafile by clicking the Playback button. Then choose Hit Test from the Edit menu. The cursor is changed to a cross hair when the mouse pointer is over the drawing surface.
- Then click the graphic object played back in the drawing surface. The editor will search through the metafile, record after record, to find the record that corresponds to the object based on the mouse position. The search provides a visual cue by changing the graphic objects to red as it goes until it hits the corresponding object. If there is a hit, the record number is displayed on the control panel, a beep is heard, and a hit message is displayed on the bottom.

When done, clear the Hit Test menu item.

Random Access Playback

- Click the number button in the control panel to play back a particular record. To access a non single-digit record, click the 10+ button an appropriate number of times and then click the appropriate number button to bring the sum to the record desired.

Playing Back Metafile Records One-by-One

- Click the Fast Forward button to play the metafile records one at a time.

Selective Recording of Existing Enhanced Metafile Records into a New Enhanced Metafile

- Click the Record button and the appropriate number button for playing back selective metafile records in the drawing surface. The playback records will be recorded into the new metafile. When done, click the Stop button.

Drawing with Pen, Text, Bezier, Line, Ellipse, Rectangle and Embedding Bitmap and Enhanced Metafile Tools

- The default pen is black. To change the pen color, choose the Pen menu item from the Options menu to select a color.
- The default brush used by the Fill Rectangle and Fill Ellipse routines is black. To change the brush color, choose Brush from the Options menu to select a color.

- The Text tool uses the default system font. To change the font, choose Font from the Options menu to change to a different font and font attributes.
- The Bezier tool takes four points initially and three thereafter. To draw a bezier curve, select the Bezier tool and click the drawing surface three or four times to place the control points.
- To embed a currently loaded bitmap, select the Bitmap tool and click three points on the drawing surface to describe where you want the bitmap located. The editor does the proper transformation on the bitmap and embeds the bitmap in the drawing.
- To load a bitmap, choose Load Bitmap from the File menu and make the selection. The Bitmap tool optionally takes a mask bitmap. The mask bitmap must be a monochrome bitmap.
- To load a mask bitmap, choose Load Mask Bitmap from the File menu to make the selection. Select a color bitmap, because the mask has the effect of resetting the mask to none.
- To embed a currently loaded enhanced metafile, select the Embed Enhanced Metafile tool and click three points on the drawing surface to describe where you want the enhanced metafile located. The editor does the proper transformation on the metafile and embeds it in the drawing.

Additional reference words: 3.10

SAMPLE: File I/O API Functions Demonstration

Article ID: Q94905

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

FILER.EXE - File I/O Sample

SUMMARY:

Filer.exe is a basic file management applet, ala File Manager or Norton CommanderTM. It demonstrates many of the new File I/O API, and their related algorithms, such as drive enumeration, .EXE Version Information retrieval and directory tree expansion. It also demonstrates many intensive user algorithms, such as child management, subclassing, synchronization and control window management.

DESCRIPTION:

Filer presents the user with two configurable child windows, each of which the user may associate with a drive from the drives available on the system.

A Drive Toolbar describes the available drives on the system. Users select drives from the toolbar or the Drives menu. A function toolbar also corresponds to the file I/O functions in the File menu. A command line window at the bottom of the app will spawn a command shell with the command given, and the option of keeping or destroying the command window after the given command completes.

Each of these Drive child windows contains a Directory ListBox, and a File ListBox, with which the user may browse through the files on the selected drive.

Filer gives the user the option to Open (execute/edit), Copy, Delete, Move, and Rename files, as well as Make and Remove Directories, and display version information embedded in Win32 files. The active Drive child acts as the source, and the inactive Drive child acts as the default destination of file I/O operations.

The Drive children may also be configured as side by side or above and below one another, and the File and Directory Listboxes in each may swap positions. The user may opt to fully expand the Directory trees.

All features of Filer may be selected from the Mouse, Keyboard, or by Menu Items.

FUTURE ENHANCEMENTS(?):

- Directory copy, move, and delete.
- Font choice.
- Online Help.
- Save configuration.
- File associations.
- Network drive functions.
- File size, last change information; view/modify file attributes.
- Total size of given single/multiple selection of files.

Sample: Demonstration of Journal Hooks Under Win32

Article ID: Q99342

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The Minimum Recorder (MinRec) sample demonstrates journal record and playback hooks in the context of a simple recorder application.

More Information:

The following is a complete list of files accompanying the Minimum Recorder sample:

MAKEFILE	MinRec makefile
MINREC.C	Main source module
MINREC.DEF	Main source module definition file
MINREC.DLG	Dialog box templates
MINREC.H	Main source module header file
MINREC.ICO	MinRec icon
MINREC.RC	Resource file
README.TXT	MinRec readme file, which you are reading now
RECHOOK.C	Hook functions and procedures in a DLL
RECHOOK.DEF	Hook DLL definition file
RECHOOK.H	Hook DLL header file

Additional reference words: 3.10 read me make file

Sample: Dynamic Dialog Box Creation

Article ID: Q99461

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

An application usually defines a dialog box template in an RC or DLG file, then creates a dialog box at run time from that. However, the dialog box template can be created at run time. The DYNDLG sample shows how to do this for Win32.

More Information:

The dialog box template in Win32 is conceptually similar to the Windows 3.1 version, but is different enough to require some porting work. For example, any strings are now stored in Unicode. The dialog box template format is documented in the associated Help file.

Run-Time Dependency

One of the dialog boxes uses a custom control that is expected to be defined in the SPINCUBE.DLL file. The DYNDLG sample attempts to load this DLL at run time in the following location

```
..\spincube\spincube.dll
```

that is, in a directory of the same level, but named SPINCUBE. This can be changed, and the sample recompiled. SPINCUBE is another sample distributed in the Win32 SDK.

Additional reference words: 3.10

SAMPLE: Win32s Universal Thunks

Article ID: Q105170

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

UTSAMPLE can be found in the MSTOOLS\WIN32S\UT\SAMPLES\UTSAMPLE directory of the Win32 SDK CD.

This sample shows how to use Universal Thunks to call application programming interfaces (APIs) that are not supported directly by Win32s.

This sample detects at run time whether it's running under Win32s or Windows NT, and checks to see whether or not the APIs it calls are supported; if they are not supported, the sample calls the appropriate 16-bit APIs via Universal Thunks.

Because this sample uses 16-bit code, you need a 16-bit compiler (not included in the Win32 SDK). The included MAKEFILE.16 uses Microsoft C/C++ version 7.0, but it could be easily adaptable to other compilers.

Also, you need to obtain some components from the \MSTOOLS\WIN32S\UT directory on your Win32 SDK CD-ROM. In particular, the W32SUT.H file needs to go in %MSTOOLS%\H and also in a directory in the INCLUDE path for your Win16 development environment. W32SUT32.LIB needs to go in %MSTOOLS%\H, and W32SUT16.LIB needs to go in a directory in the LIB path for your Win16 development environment.

Additional reference words: 3.10

Sample: How to Reboot or Shut Down Programmatically

Article ID: Q95966

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The BOB (big orange button) sample demonstrates the steps necessary to reboot or shut down the machine in Windows NT and Win32s/MS-DOS programmatically. A similar method can be used to log off a user. The steps are as follows:

1. Check to see if the machine is running Windows NT; if so:
 - a. Get the security token.
 - b. Fetch the LUID for the SeShutdownPrivilege; this is required only for rebooting and shutting down the machine.
 - c. Enable the shutdown privilege.
2. Use ExitWindowsEx() to log off, shut down or reboot the machine.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory; small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

CAUTION: Use this API (application programming interface) with care *and* with plenty of warning so that a user is not rudely surprised.

There are two methods to prevent the system from shutting down, one at the application level and one at the system level. At the application level, process the WM_QUERYENDSESSION message so the application can ask the user whether he or she wants to save any data that has not been backed up to disk. Then, return TRUE assuming the process can shut down safely or FALSE if it cannot [note: this will not keep the machine from rebooting or shutting down if the parameter to ExitWindowsEx() was or'ed with EXW_FORCE]. At the system level, it is possible to remove the rights of certain users/groups to reboot the machine. This is done via the UserManager application.

Additional reference words: 3.10

Sample: Creating a WinDbg Extension

Article ID: Q95967

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The purpose of the WDBGEXTS sample is to demonstrate how to create a WinDbg (WinDebug) extension. This is a port of a sample demonstrating how to create an NTSD extension.

Extensions are dynamic-link library (DLL) entry points. The arguments passed to an extension are:

HANDLE hCurrentProcess - Supplies a handle to the current process (at the time the extension is called).

HANDLE hCurrentThread - Supplies a handle to the current thread (at the time the extension is called).

DWORD CurrentPc - Supplies the current pc (program counter) at the time the extension is called.

PWINDBG_EXTENSION_APIS lpExtensionApis - Supplies the address of the functions callable by this extension.

LPSTR lpArgumentString - Supplies the command-line arguments for the extension.

The type PWINDBG_EXTENSION_APIS is defined in \MSTOOLS\H\WDBGEXTS.H.

Note that in the makefile, the -Gz option is specified to the compiler in order to ensure that __stdcall is used.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory; small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

The following is a description of the exported functions:

Function	Description
-----	-----
igrep()	Searches the instruction stream for a pattern.
str()	Given a pointer to a string, it prints out the string, its length, and its location in memory.

To use the commands contained in WDBGEXTS.DLL, make sure that the DLL

is placed in a directory that is on the path.

The syntax for the commands is as follows:

```
!wdbgexts.igrep [pattern [expression] ]
```

```
!wdbgexts.str [string]
```

Additional reference words: 3.10

Sample: DDEML API Demonstration

Article ID: Q96395

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Install SDK sample provides an example of how to use the DDEML API's to add groups and program items to the Program Manager. The program can be run either in interactive or batch mode and the search path can be specified on the command line.

More Information:

The program can be started from the command line. The following flags can be used.

(-/) (sS) <path> Specify the search path. Currently this must be single path. The application searches for an environmental variable "MSTOOLS". If this variable is not found then the default value is c:\mstools\samples.

(-/)g(G) <name> Specify the name of the group that the items should be added to. The default value is 'Sample Applications'.

(-/)b(B) Specifies that the program should run in batch mode. The program will find all of the .exe files in the specified path, searching recursively. It will then create the specified group and add all of the found executables to the group. The program will then exit.

(-/) (iI) Specifies interactive mode. This is the default. The program will find all of the exe files in the specified path, searching recursively. The names will be displayed in a list with the name that will be use displayed in an additional list. A drop down combo box will display a list of the currently available groups. The user can type in a new group name if desired.

If the program finds more than 50 executables then after adding 50 items to a group a new group is created using the name currently selected in the Program Manager Group list with "Part <n>" appended where <n> starts at 2 and is incremented after every new group is added.

Sample: Demonstration of the Win32 Debug API

Article ID: Q96396

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Debug Event Browser (DEB) Win32 SDK sample demonstrates the new Win32 Debug API. This sample acts as a debugger for both newly created debuggee processes or attaches to current active processes.

DEB is not a debugger in the traditional sense; it is a browser as its name implies. DEB displays the debug events, and their relevant properties, as they occur and invokes the default handlers supplied either by the debuggee or the system. Only the minimal debug event handling is imposed such that the debug events are displayed and the debuggee is continued on its normal course of execution.

The sample is source code compatible for both the Intel 80x86 and MIPS R4000 Windows NT platforms.

More Information:

The following is a complete list of files accompanying the Debug Event Browser sample:

DEB.BMP	DEB bitmap used by DEB.RTF
DEB.DEF	DEB module definition file
DEB.DLG	DEB dialog resource script file
DEB.H	DEB ID values and user message defines
DEB.HPJ	DEB help project file
DEB.ICO	DEB main icon
DEB1.ICO	DEB animated icon number one
...	
DEB8.ICO	DEB animated icon number eight
DEB.RC	DEB resource file
DEB.RTF	DEB help topic file
DEBDEBUG.C	DEB debug support functions
DEBDEBUG.H	DEB debug support functions header
DEBMAIN.C	DEB main module - WinMain and callbacks
DEBMAIN.H	DEB main module header
DEBMISC.C	DEB miscellaneous support functions
DEBMISC.H	DEB miscellaneous support functions header
LINKLIST.C	Ordered doubled-linked list library
LINKLIST.H	Ordered doubled-linked list library header
MAKEFILE	Make/Nmake file for the entire sample
README.TXT	This file you are presently reading
TOOLBAR.BMP	Toolbar bitmap used by DEB.RTF
TOOLBAR.C	Toolbar functions

TOOLBAR.DEF Toolbar module definition file

Sample: GUIGREP File Manager Extension Sample

Article ID: Q96397

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The GUIGREP SDK sample implements a File Manager extension that can serve as a source-file browser. Similar to the GREP utility, GUIGREP finds occurrences of a search string in files that can be selected through the File Manager.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

The sample demonstrates multithreading, structured exception handling, C++ code, File Manager extension functionalities, and memory-mapped file manipulation. A very small application (NTGREP.EXE) "installs" the File Manager extension by adding an entry to WINFILE.INI.

Additional reference words: 3.10

Sample: Demonstration of Using GetLocaleInfoW
Article ID: Q96398

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The LOCALE sample is a simple dialog box which provides an interface between the user and the system via the GetLocaleInfoW API.

Sample: Multiple Document Interface Demonstration

Article ID: Q96399

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MDI sample demonstrates Multiple Document Interface and associating data to each MDI window, and accelerators.

Sample: How to Share Memory Between Processes

Article ID: Q96400

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MEMORY sample demonstrates the use of the file-mapping application programming interfaces (APIs) to create a share memory between processes.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

Server

The "server" sets up the named share memory, and can be created by choosing Server from the Create menu. This opens up a multiple document interface (MDI) server child and swaps the menu bar to the server menu bar. The server menu bar includes three choices: Create, Server, and Window.

To set up the equivalent of a named share memory:

1. From the Server menu, choose Create File. This brings up the File Type dialog box. The user can create either a memory page file or a physical file by choosing the Page File or the Map File button, respectively, to back up the soon-to-be-created named share memory.

If the Map File button is chosen, the named share memory will be backed up by a physical file on the disk. If the Page File button is chosen, the memory will be backed up by the memory page file.

2. From the Server menu, choose Create File Mapping. This brings up the Map Name dialog box. The user then can specify a name for the memory map file object that is created for the file created in step 1 above.

This name is used to identify the share memory by the clients in the other processes.

3. From the Server menu, choose Map View Of File. This essentially maps the map file object created in step 2 above into the process's address space.
4. From the Server menu, choose Access. This creates a multiline edit control (MLE) inside the MDI server child. Whatever is written in

the MLE is put in the map file object.

Client

The "client" connects to the named share memory created by the server in another process. A client can be created by choosing Client from the Create menu. This opens up an MDI client child and swaps the menu bar to the client menu bar. The client menu bar includes three choices: Create, Client, and Window.

To set up the connection to the named share memory:

1. From the Client menu, choose Open File Mapping. This brings up the Map Name dialog box. The user can then enter the name of the map file object that the client wanted to connect to.
2. From the Client menu, choose Map View Of File. This essentially maps the map file object opened in step 1 above into the process's address space.
3. From the Client menu, choose Access. This creates an MLE inside the MDI Server child. Whatever was written in the map file object by the server will be shown in this MLE.

The client synchronizes with the server at a regular interval.

4. From the Client menu, choose Refresh Now to refresh the contents of the map file object.

Additional reference words: 3.10

Sample: Demonstrating the Creation of Multiple Threads

Article ID: Q96401

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MULTITHRD sample demonstrates thread creation.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

The MULTITHRD sample is a multiple document interface (MDI) application. When an MDI child is created, a drawing thread is also created with it. The drawing thread merely draws color boxes randomly inside the MDI child window.

Additional reference words: 3.10

Sample: Constructing and Using a Message Table Resource
Article ID: Q96402

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The MSGTABLE sample demonstrates how to construct and use a message table resource in your application. The sample message table (MESSAGES.MC) for the message compiler (MC.EXE) shows the format of a typical message table source file. Note that some of the entries for each message are optional but are shown for demonstration purposes. See the documentation in the TOOLS.HLP file on the message table compiler for more information on which fields are optional.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

This sample shows how to create a resource-only dynamic-link library (DLL) that contains nothing but the message table resource. The DLL source file (MESSAGES.C) has a stub entry point that merely returns TRUE. This is due to the requirement that a DLL have at least one entry point, so we need to have one for our resource-only DLL.

The MSGTEST.EXE executable shows how to load the message table DLL, how to extract the message text from the resource with the FormatMessage API, and how to decode the various bits in the message ID. Note that the MESSAGES.H include file to the executable is created during the make process by the message compiler (MC.EXE) as it is creating the binary resource file from the message table source file.

The makefile should be useful to demonstrate how to set up your dependency rules for your message table source files and message table DLL.

This sample could be easily modified to link the message table resource directly into your executable rather than into a stand-alone DLL; there is no requirement that the message table resource be located only in a DLL, although this is the common case.

Additional reference words: 3.10

Sample: Sharing Named Memory Between Two Processes
Article ID: Q96403

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

SHAREMEM and OTHRPROC are two samples which work together to demonstrate sharing named memory between two separate processes. These samples must be run together.

More Information:

To use: first start an instance of SHAREMEM. A window will appear, divided into an upper and lower section. The upper section will have two edit fields: one displaying the mouse pointer's X-coordinates, and the other the pointer's Y coordinates, as the mouse moves in the lower section.

Start an instance of OTHRPROC. Allow OTHRPROC to keep focus, but move the pointer around over the SHAREMEM'S window. OTHRPROC has a similar window configuration; however, you will notice that the X and Y coordinates of the mouse as it moves over the SHAREMEM window are the values that appear in the edit fields of OTHRPROC. To emphasize this, a cross hair will appear in OTHRPROC's lower section; its movements relative to the mouse position in SHAREMEM's window.

What's happening: When SHAREMEM is started, it creates and allocates a piece of named shared memory the size of a DWORD (the size needed to hold the mouse cursor's X and Y coordinates) using CreateFileMapping. As the mouse pointer is moved across the window, the WM_MOUSEMOVE messages are trapped, and the coordinates are written to the upper edit fields and to the piece of named shared memory.

When OTHRPROC is started, it gets access to the named shared memory by calling OpenFileMapping and MapViewOfFile. OTHRPROC then uses a thread to poll and read the X and Y coordinates written to the shared memory by SHAREMEM. It captures the coordinates and draws a bit map of a cross hair in the specified location.

Additional reference words: 3.10 crosshair

Sample: Platform Detection

Article ID: Q96404

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

All Win32 applications need to use `GetVersion()` to determine whether they're on the Win32s or Win32 platform. `GetVersion()` sets the high bit of the high word if the application is running on Win32s.

Win16 applications can determine whether there are running on Win32/NT or Win16/MS-DOS by examining the flags return by `GetWinFlags()`; a unique manifest constant is returned if the application is running in the Windows on Windows (WOW) layer.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the `\MSTOOLS\SAMPLES` directory, and small samples available on the CD in the `\Q_A\SAMPLES` directory.

Additional reference words: 3.10

Sample: Demonstration of Printing with Windows NT

Article ID: Q96405

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The PRINTER sample does the following:

- Shows how to print on Windows NT, using both the `CreateDC()` and the `PrinterDlg()` methods for acquiring a printer HDC. The user is allowed to print different graphics objects, as well as a complete device font set. An Abort dialog box is also implemented.
- Provides complete device capabilities for all printers and the display.
- Provides information (levels 1 and 2) returned by a call to `EnumPrinters()`.
- Shows how to enumerate fonts for a particular DC.
- Illustrates differences between the various mapping modes.
- Demonstrates GDI functionality.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the `\MSTOOLS\SAMPLES` directory, and small samples available on the CD in the `\Q_A\SAMPLES` directory.

More Information:

The main application window contains a menu and a toolbar. The various submenus allow for:

Submenu -----	Description -----
Print	Calls <code>CreateDC</code> to get a device context for the selected printer in the toolbar combo box, and then prints the current graphics options to this DC.
PrintDlg	Calls <code>PrintDlg</code> to retrieve a device context for a printer, then prints out current graphics options to this DC.
GetDeviceCaps	Retrieves device capabilities for the device currently selected in the toolbar combo box, and displays them in a dialog box.

EnumPrinters	Retrieves levels 1 and 2 information returned by EnumPrinters, and displays this information in a dialog box.
GetPrinterDriver	Returns levels 1 and 2 information returned by GetPrinterDriver (for the currently selected printer) and displays this information in a dialog box.
EnumPrinterDrivers	Returns levels 1 and 2 information returned by EnumPrinterDrivers, and displays this information in a dialog box.
Refresh	Refreshes the contents of the toolbar combo box (changes made in Print Manager are reflected by this).
About	Application information dialog box.
Mapping Modes	User chooses between different mapping modes.
Graphics	User chooses different primitives to display.
Pen	User can configure size, color, and style of the drawing pen.
Brush	User can configure size, color, and style of the drawing brush.
Text color	User can configure color used to draw fonts.

Additional reference words: 3.10 combobox

Sample: Named Pipe Client/Server Demonstration

Article ID: Q96406

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

NPCLIENT and NPSEVER demonstrate the use of named pipes. The basic design consist of a server application serving multiple client applications. The user can use the client applications as an interface to all of the other client applications via the server. The effect is a simple communication program that can be used over the network between multiple clients.

More Information:

The actual implementation works by having the NPSEVER application launch a new thread, which creates and services a new instance of the server side of the named pipe every time a client connects to it. You need only start one instance of the NPSEVER application. It will service up to 100 instances of the NPCLIENT application. (Note that the 100 instance limit is hard coded into the sample. It does not reflect the number of named pipe instances you can create, which is virtually infinite.)

TO USE:

Start an instance of NPSEVER. A window will appear.

Start an instance of NPCLIENT. Two dialog boxes will appear, one on top of the other. The top level dialog box will prompt you for a share name and a client or user name. If the instance of NPCLIENT is local to (on the same machine as) the NPSEVER instance, enter a '.' for the share name. Otherwise, enter the machine name of the server that the NPSEVER instance was started on, i.e. 'FoobarServer'. For the client or user name, enter any name you wish to be identified with. Hit enter or click the OK button.

The upper dialog box will go away, and you'll see the Client dialog box of NPCLIENT. It consists of two edit fields and a 'Send' button. You will be able to read messages from other clients (and yourself) in the larger/upper edit field. (Note, if the message seems garbled, make sure the cursor of the edit field is located in the lower left hand corner of the field.) The smaller edit field is used to type messages. To send a message: type something in the lower/smaller edit field, and hit enter or click the Send button. The message will appear in the larger edit field of all the clients connected to the NPSEVER instance;

prepending by the user name you selected. Note that the user name you selected will be entered into the caption bar of the NPCLIENT instance. This allows you to more easily keep track of multiple instances of NPCLIENT on the same machine.

At the same time the top level dialog box was dismissed from the NPCLIENT instance, the NPSEVER window was updated with the picture of a red spool of thread accompanied by the user name you selected. This red spool indicates an active client thread connected to NPSEVER. The spool may be connected to other spools with a thin blue line (similar to the way the File Manager connects files or directories). Any time a client disconnects from NPSEVER; the spool representing it will be grayed out.

DESIGN:

Basically, the NPSEVER application launches multiple instances of a server thread. When the application is started, the first thread is created. It creates an instance of the server side of the named pipe, and waits for a client to connect. Once a client connects, another thread is started and it too blocks waiting for a client. Meanwhile, the first thread updates a global array of client information with this specific client's information. The thread then enters a loop reading from this client. Any time this specific client sends a message, this server thread will call a function (TellAll) which will write the message to all the clients that have been listed in the global array.

On the client side, NPCLIENT tries to connect to the named pipe with a CreateFile call. Once it has connected, it creates a thread which loops and reads any message from the server side. Once a message is read, it is printed in the larger edit field. Any time the user hits the Send button, the main thread grabs any text in the lower edit field, and writes it to the server.

The steps between NPSEVER and an instance of NPCLIENT looks like this:

NPSEVER	NPCLIENT
-----	-----
CreateNamedPipe()	
ConnectPipe() // Blocks	
	CreateFile() //Connects to pipe. spawn separate thread to read pipe
return from block	
updates array of clients	
spawn another server thread	
Loop	
ReadFile() // Blocks on overlap	WriteFile() // User hits Send.
return from block	

```
WriteFile() // Broadcast to clients
End loop    // When client breaks pipe.
```

```
ReadPipe Thread:
```

```
Loop
```

```
ReadFile()
```

```
    block till server broadcasts
```

```
    return from block.
```

```
    put string in edit field.
```

```
End loop // when server breaks.
```

The overlapped structure should be used anytime a pipe is expected to block for any length of time on a read or write. This allows the thread to return immediately from a read or write to service any other part of your application. The overlapped structure should also be used on a named pipe anytime you expect to do simultaneous reads and writes.

Sample: Read/Write Synchronization Demonstration

Article ID: Q96407

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

This article refers to the file DATABASE.C, a part of the READWRIT sample, which is one variation of the classical synchronization problem, Reader/Writer. The Reader/Writer problem, first stated and solved by Courtois, involves a shared resource--typically a database. The issue is to allow both readers and writers access to the database without corrupting it.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

Multiple readers are allowed to access the database as long as a writer is not accessing it. However, when a writer is accessing the database, no other readers or writers are allowed access. There are several variations of this problem; the simplest favors readers to the exclusion of writers, and visa versa.

The implementation used by the READWRIT sample allows multiple readers in the database at one time until a writer wants entrance. Then, no other reader can enter the database until this writer is finished. All readers currently in the database, however, can finish. Thus, no starvation of either the readers or writers, which is inherent in simpler methods, will happen.

To keep this sample focused, the user interface is very simple. To run this sample, type the following at the command prompt:

```
READWRIT
```

You will see the values the readers put on the screen. These are a running total of the threads that wrote to the database.

Additional reference words: 3.10

Sample: Using API Functions to Access the Registry
Article ID: Q96408

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The Registry Monkey is a simple utility which demonstrates the Registry API functions needed to access the NT Registry. Monkey can be used to climb up and down the various branches of the Registry tree, displaying the individual key's data values. Monkey can also be used to print specified trees to a file named Registry.txt.

The Registry Monkey sample can be found in the More Information:

To use: start an instance of the Monkey. A dialog box will appear with several edit fields, list boxes and buttons. The listbox in the center of the dialog box (labeled CHILD KEYS: at the bottom) will always hold the child keys of the current key. Initially it has four entries, representing the four pre-defined key handles of the Registry: HKEY_LOCAL_MACHINE, HKEY_CURRENT_USER, HKEY_USERS, and HKEY_CLASSES_ROOT. If you double click on any of these entries, or high light it and press the "Next/Down" button; the key that you just selected will appear in the edit field "Key Name", and the children of that selected key will replace the entries in "CHILD KEYS:" list box. I.e. if you select HKEY_LOCAL_MACHINE, that name will be present in "Key Name", and it's children will appear in the list box: HARDWARE, SECURITY, SOFTWARE, and SYSTEM. To proceed deeper into the tree, double click another child. To back out of the Registry, double click on the ".." at the top of the listbox, or press the "Back/Up" button.

If the current key has values associated with it, the name of the values will be listed in the right hand listbox (labeled "VALUES:"). If it has now values, "VALUES:" will be followed by a "0". Once you come upon a key that does have values associated with it, you can double click on any of the values in this list box. At the bottom of the dialog box are two edit fields: "Value: Data Type", and "Value: Data Entry". By double clicking a value entry, these edit fields will be filled in the data's type and the data's value. I.e. if you follow the tree down to HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System; and double click on the entry "1) Identifier", the "Value: Data Type" field will be filled with "REG_SZ: A null-terminated Unicode string"; and the "Value: Data Entry" may be filled in with something such as "AT/AT COMPATIBLE"

If the current key has a Class type associated with it, it will appear in the "Class" edit field. The "ACL" edit field is not implemented with this release of the Registry Monkey.

You can use the Registry Monkey to write any part of the Registry Tree to a file called REGISTRY.TXT. To do this, select either the "Full" or "Trimmed" buttons (this specifies either writing all of the key entries, or only those having Value data associated with them); and press the "Print Branch" button. The Registry Monkey will begin at the current branch, and will proceed recursively down the branches to the end of the tree, writing the information to the file. To write the entire tree, print each of the four pre-defined keys. Note, this can make for a rather large file (700Kb at the time this was written).

Sample: Fractal Screen Saver Demonstration

Article ID: Q96409

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The FRACTAL screen saver sample draws the Mandelbrot Set on the screen, then saves the picture in the path as specified in the configuration dialog box. The default path is C:\FRACTAL.BMP.

When it is done, the screen saver animates the palette (for palette-manageable devices only.)

Users can specify the area of the Mandelbrot Set to draw as well as the number of iterations.

To install the screen saver, copy the executable FRACTAL.SCR to the .\NT\SYSTEM32 directory. Then install the screen saver in Control Panel.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

Additional reference words: 3.10

Sample: Using Semaphores to Control Threads

Article ID: Q96410

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The SEMAPHOR sample application shows how to control four threads with a semaphore. It demonstrates this by having four threads competing for the right to draw their color to a rectangle in the center of the window. Access to the center area is controlled by the semaphore.

More Information:

When you start the application you will see five rectangles: a dynamic rectangle in the center (always changing color), and four static rectangles surrounding it. Each of the four rectangles has its own color: red, blue, green, and gray. The one in the center alternates between these colors.

The four static rectangles represent four threads. These four threads compete for the rectangle in the middle, and their access is controlled by the semaphore. When a thread gains control of the semaphore, it gets to draw its color in the center rectangle. (Note: The threads do not actually draw any of the four static rectangles. To make the code simpler, this is handled in the WM_PAINT message in the MainWndProc function. The rectangles are used only as visual representations of the threads. The threads do, however, draw the rectangle in the center with their specific color.)

The semaphore has a use count. When it is set to zero, any thread can access the semaphore and execute the code within its "semaphore gate" by using WaitForSingleObject. When the thread gains control of the semaphore using this call, the use count is incremented by 1. When the thread is done executing its code, it can call ReleaseSemaphore. This will decrement the count by whatever value you indicate (this code uses 1), signaling to any other thread that it may gain control of the semaphore.

Note: Any thread that has access to the semaphore may decrement the semaphore's use count with ReleaseSemaphore; the thread does not have to have control of the semaphore at the time.

In this code, the WM_CREATE message in MainWndProc creates a semaphore. The four threads are then created, each waiting on the semaphore.

Each of the threads loop, blocking on a WaitForSingleObject call. Once any thread has set the use count to 0, the thread gets to draw the center rectangle with its color and then sleeps for half a second before freeing the semaphore again. The thread then runs through the loop again.

Additional reference words: CreateSemaphore WaitForSingleObject ReleaseSemaphore 3.10

Sample: Demonstration of Opening and Terminating a Process
Article ID: Q96412

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

This simple piece of code demonstrates how to open and terminate a process in situations where it is impossible to do so otherwise. PSTAT.EXE can be used to obtain the process ID (PID) of the process to be terminated. Note that the application does the decimal-hexadecimal conversion as is necessary.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

Additional reference words: dec-hex

Sample: Using TLS to Store Thread-Specific Data in a DLL
Article ID: Q96413

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The TLS sample demonstrates using thread local storage (TLS) to store thread-specific data in a dynamic-link library (DLL). As each thread attaches, the memory is alloc'd and filled. As each thread detaches, it is retrieved; as the process detaches, the TLS is freed.

This is a very basic sample, and the calls into the DLL that cause TLS to be allocated and filled don't really do anything; however, this should be a good basis on which to build TLS.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

Additional reference words: 3.10

Sample: World Coordinate Transforms

Article ID: Q96414

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The WORLD sample demonstrates how an image can be translated and scaled using `SetWorldTransform()`. The image is read from the metafile.

After the application is started, choose Open from the Metafile menu to specify which metafile should be used. The image is displayed, and the horizontal and vertical scroll boxes ("thumbs") are set to the middle of the scroll bars. Use the scroll bars to translate the image. To scale the image, select the Scale menu option and choose the desired horizontal and vertical scaling factors from the list boxes.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. Samples are divided into two groups: samples (optionally) installed into the `\MSTOOLS\SAMPLES` directory, and small samples available on the CD in the `\Q_A\SAMPLES` directory.

More Information:

The sample metafile `SAMPLE.EMF` was created using code from the Win32 "Programmer's Reference: Overviews" manual, Chapter 74, "Metafiles." `PlayEnhMetafile()` is used in the `WM_PAINT` case to display the image stored in the metafile.

`SetWorldTransform()` takes two parameters: a device context and a structure that defines the transform to be applied. The transform contains six fields, which have the following use in this program:

```
eM11 : Horizontal scaling factor
eM12 : Not used
eM21 : Not used
eM22 : Vertical scaling factor
eDx  : Horizontal translation
eDy  : Vertical translation
```

The `WM_HSCROLL` and `WM_VSCROLL` cases are handled like they are handled in any of the other samples, except that `ScrollWindow()` is not used to update the window contents. Instead, the appropriate modification is made to the transform, and `InvalidRect()` is called to update the window.

When the OK button is chosen in the Scale Image dialog box, the transform is updated and `InvalidRect()` is called to update the window.

Additional reference words: 3.10 scrollbar listbox

SAMPLE: Primitive Drag and Drop Unicode Input Method

Article ID: Q96415

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The UNIPUT sample provides a primitive, mouse-based input method. It allows the user to grab any character covered by a Unicode font, and to drag the character to a second window. If the user drops the character on the second window, that window is sent a WM_CHAR message with the appropriate key code.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. Samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

In the UNIPUT sample, the status bar at the bottom of the window shows three fields of information: the title of the last window to receive a WM_CHAR message is on the left; the type of that window, either Unicode or ANSI, is in the center; a list of the most recently dropped characters, sort of a history buffer, is on the right.

There is an online help file containing more information.

Notice:

There is a font distributed on the Win32 SDK disk which covers more than one thousand unicode characters. In order to use this application to its full potential, you will want to install that font using the Control Panel, Fonts item. The font is named L_10646.TTF.

Additional reference words: 3.10

Sample: WINDIFF Source Included as an SDK Sample
Article ID: Q97655

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

WINDIFF compares directories or files giving a graphic comparison.

Full command-line syntax:

```
WINDIFF [paths] [saveoption]
```

[paths]

path

To compare what is at path with what is in the current directory.

path1 path2

To compare what is at path1 with what is at path2.

Note: Path can always be relative or absolute, net or local, file or directory.

[saveoption] = -slrd savefile

Where slrd is any combination of these four letters (s, l, r, d) to write the names of files that are:

- s - The same in both paths.
- l - Only in the left-hand path.
- r - Only in the right-hand path.
- d - In both paths, but the two files are different.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. Samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

More Information:

The following information summarizes WINDIFF's use of color and its menu items:

[Colors]

- RED background designates text from left file.
- YELLOW background designates text from right file.
- BLUE text designates a moved line.
- BLACK text is used for everything else.

[File menu]

- Compare files...
Uses the File Open dialog box for each of two files to be compared.
- Compare directories
Opens a dialog box to allow entry of two directory names for comparison.
- Close
Closes the current files.
- Abort
Will be unavailable (grayed) unless an operation is in progress; then, allows that operation to be terminated before completion.
- Save File List
Allows the list of files that are (the same, different, only in left or only in right) to be saved.
- Copy Files...
Opens a dialog box allowing you to write the files to a disk.
- Print
Prints out the results of the compare.

[View menu]

- Outline
Show lists of files.
- Expand
Show comparison of selected files.
- Picture
Show picture as well as selected files.
- Previous Change (F7)
Skip to previous point of difference in the file.
- Next change (F8)
Skip to next point of difference in the file.

[Expand menu]

- Left file only
Show only lines from left file (but colored so as to highlight changed lines).
- Right file only
Show only lines from right file (but colored so as to highlight changed lines).
- Both files (default)
Show a merge of both files. All the lines in the left file are shown in the order in which they occur in that file, likewise for the right file. Lines that are ONLY in the left file are shown in red. Lines that are ONLY in the right file are shown in yellow.
- Left line numbers
Line numbers are shown, based on the left file.
- Right line numbers
Line numbers are shown, based on the right file.
- Right line numbers
Line numbers are shown, based on the right file.
- No line numbers
Line numbers are turned off.

[Options menu]

Ignore blanks

Blanks are ignored in the expanded view, so that lines that differ only in white space are shown as identical.

Show Identical Files

Include files that are identical in each path in outline mode.

Show Left-Only Files

Include files that occur only in the left path in outline mode.

Show Right-Only Files

Include files that occur only in the right path in outline mode.

Show Different Files

Include files that occur in both paths, but which are not the same in outline mode.

[Help]

About

Displays version information about WINDIFF.

Additional reference words: 3.10

SAMPLE: Monitoring System Events

Article ID: Q97656

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

SUMMARY

=====

This sample uses a variety of thread-specific hook procedures to monitor the system for events affecting a thread. The sample demonstrates how to process events for the following types of hook procedures:

- WH_CALLWNDPROC
- WH_CBT
- WH_DEBUG
- WH_GETMESSAGE
- WH_KEYBOARD
- WH_MOUSE
- WH_MSGFILTER

HOOKS is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. Samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

MORE INFORMATION

=====

In this sample, the user can install and remove a hook procedure by using the menu. When a hook procedure is installed and an event that is monitored by the procedure occurs, the procedure writes information about the event to the client area of the application's main window.

Additional reference words: 3.10

KBCategory:

KBSubcategory:

SAMPLE: Examining Security Descriptors (SDs)

Article ID: Q97657

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

In the Win32 .HLP file, if you click Search, then "Security Overview," and then from the list of topics under Security Overview choose the subtopic "Allowing Access," you will find the comment:

Note: It is fine to write code like this that builds security descriptors from scratch. It is, however, a good practice for people who write code that builds or manipulates security descriptors to first write code that explores the default security descriptors that Windows NT places on objects. For example, if Windows NT by default includes in a DACL an ACE granting the Local Logon SID certain access, it's good to know that, so that a decision not to grant any access to the Local Logon SID would be a conscious decision.

Purpose of This Sample

The comment in the .HLP file is accurate; however, for many people the task of examining the security descriptor (SD) is easier if there is sample code to start from. Therefore, the purpose of this sample is to provide sample code for use as a starting point from which to examine SD(s). This sample examines the SD on files; this code can be modified to examine the SD on other objects.

This sample is not a supported utility.

To Run the Sample

Type `Check_sd` to check the SD on the `\\.\A:` device.

Type `Check_sd d:\a.fil` to check the SD on the `d:\a.fil` file. In this case, drive D must be formatted NTFS, because only NTFS files have SD(s).

Further Notes

- If you recompile with the define set as follows

```
#define I_DO_NOT_WANT_THIS_CODE_TO_CLUTTER_THIS_PROGRAM_S_OUTPUT (1==0)
```

and re-run the program, the program will produce more output, including displaying all the information you can access in a Win32

program from the process's Access Token, and the SDs of some sample objects.

- If you log on, run with the program built to produce the extra output as mentioned just above, save that output to a file, then log off and re-run the program, save the output of this second run to a different file. With WinDiff, you can easily observe how the local logon SID really does change values for each logged on session.
- A sample test you could run to exercise DACLs involves using the \Q_A\SAMPLES\SD_FLPPY sample in conjunction with this check_sd sample.
- Perform the following series of steps:
 1. Log on to a machine as a local Administrator.
 2. Do the following:

```
check_sd \\.\A: >out_bef.a
check_sd \\.\B: >out_bef.b
```
 3. Log off.
 4. Log on to the same machine as Guest on the local machine domain.
 5. Do the following:

```
sd_flppy
```
 6. Try the following:

```
dir a:      (observe access denied)
dir b:      (observe access denied)
copy config.sys a:\  (get device not found)
copy config.sys b:\  (get device not found)
```
 7. Log off.
 8. Log on to the same machine as a local Administrator.
 9. Do the following:

```
check_sd \\.\A: >out_aft.a
check_sd \\.\B: >out_aft.b
```
 10. Browse the differences between OUT_BEFL.* and OUT_AFT.*
- The above sample test demonstrates that the ACLs that sd_flppy applies survive logoffs. To demonstrate the DACLs do not survive rebooting, simply reboot, log back on as a local Administrator, and

```
check_sd \\.\A: >out_rbt.a
check_sd \\.\B: >out_rbt.b
```

to see the DACLs are again as they were in the following:

out_bef.a
out_bef.b

Additional reference words: 3.10

Sample: Using Based Pointers to Share Memory

Article ID: Q97658

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

SUMMARY

=====

When several processes need to access shared data, there is no guarantee that the shared memory is mapped to the same locations in all processes. This can cause a problem when the data contains relative pointers because a pointer value that is valid in one process's context may not be valid in the context of the other processes.

BPOINTER is a sample that demonstrates the use of based pointers to allow manipulation of shared data from several processes using memory mapped files. This technique is applicable to all forms of shared memory.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. Samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

MORE INFORMATION

=====

The sample contains the following modules:

READDATA.EXE	A console process that allows you to view the shared data; it dereferences pointers as it encounters them.
CHGDATA.EXE	A console process that lets you add elements to a shared linked list.

Note, however, that based pointers reduce the performance of the application using the pointers because the pointers must be resolved at run time; that is, each access typically adds one machine instruction overhead when dereferencing a pointer.

Once the files are compiled, execute CHGDATA.EXE and follow the instructions posted there.

Additional reference words: 3.10

SAMPLE: Event Logging

Article ID: Q98613

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT version 3.1
-

Summary:

The sample called LOGGING shows how to do event logging. The sample demonstrates how to set up the proper registry entries to register your message dynamic-link library (DLL), how to make the actual event-log entries, and how to pull event-log data from the event log.

This sample is included with the Microsoft Win32 Software Development Kit (SDK) for Windows NT. The samples are divided into two groups: samples (optionally) installed into the \MSTOOLS\SAMPLES directory, and small samples available on the CD in the \Q_A\SAMPLES directory.

Additional reference words: 3.10

Sample: Combo Boxes and Owner-Draw Techniques

Article ID: Q99462

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The OWNCOMBO sample application illustrates the use of functions and messages for combo boxes and owner-draw techniques. The user interface of this sample application is self-explanatory, except perhaps for the dialog box that is created in response to the "Drop Down Combo Box" menu item. This readme file provides an explanation of this portion of the sample's user interface, to help you better interpret the source code.

When using this dialog box, the buttons send various messages to the combo box and the edit control. These buttons allow the user to vary the data sent with each message.

The following actions are performed by the buttons:

- Unsl All: This button clears (unselects) any selection in the combo box.
- Sel No: This button takes an integer value from the edit control and attempts to select an entry in the combo box given this index value.
- Sel Txt: This button takes a text string from the edit control and attempts to select the item with the given text prefix.
- Find Txt: This button searches for the text given in the edit control and returns the item number where it was found in the combo box.
- Cb Dir: This button appends a directory listing of the current directory into the combo box.
- Clr It: This button clears all the items in the combo box.
- Add It: This button takes the string given in the edit control and adds it to the combo box.
- Del It: This button deletes the currently selected item from the combo box.
- Cpy It: This button copies the currently selected item in the combo box to the edit control.

Combo Notifications Check Box:

When this box is checked, a message box appears showing what notification codes the combo box is returning to the application in response to the messages sent by the buttons.

Additional reference words: 3.10

SAMPLE: Demonstration of the Windows Sockets API

Article ID: Q99463

The information in this article applies to:

- Microsoft Win32 Software Development Kit (SDK) for Windows NT, version 3.1
-

Summary:

The WSOCK sample demonstrates the basics of sockets programming, specifically for Windows Sockets. It demonstrates how to accept incoming connections (via the Windows Sockets Asynchronous Extension APIs, threads, and traditional BSD-style blocking calls) and how to connect to remote hosts. Once connected, the user can send a text string to the remote host. WSOCK also allows the user to view information on a user-entered host name.

More Information:

For the program to operate correctly, the TCP/IP protocol must be properly installed. Also, if two machines are used over a network, both machines must have a "HOSTS" text file (for Windows NT machines, this file is located in %SYSTEMROOT%\SYSTEM32\DRIVERS\ETC\HOSTS; if TCP/IP is installed onto a Windows for Workgroups machine, the HOSTS file is located in C:/WINDOWS/HOSTS). Within each HOSTS file, both the remote and local addresses of both machines must be listed.

WSOCK can run on a single machine (execute two copies of WSOCK) or over a network with two Win32 machines. The following example explains how two separate machines over a network would test WSOCK:

1. Machine "Bob" executes a copy of WSOCK.
2. Machine "Fred" executes a copy of WSOCK.
3. Machine "Bob" chooses one of the Listen menu options (under WinSock) [Listen (Blocking), Listen With Threads, or Async Listen].
4. Machine "Fred" selects the Connect menu option (under WinSock).
5. Machine "Bob" enters "12" as a TCP port number.
6. Machine "Bob" waits for a connection.
7. Machine "Fred" enters "Bob" as the host name to connect to.
8. Machine "Fred" enters "12" as a TCP port number.

Both machines are now connected and can send strings back and forth by using the WinSock Send Message menu option.

If "Bob" exits WSOCK while there is a connection, "Fred" will receive a message box notification.

Windows Sockets calls used:

- accept
- closesocket
- connect
- gethostbyname
- getservbyname
- htons
- listen
- send
- recv
- WSAAsyncSelect
- WSACleanup
- WSAStartup

Additional reference words: 3.1

