Legal Information

Programmer's Guide to Microsoft® Windows® 95

Introduction

The *Programmer's Guide to Microsoft® Windows® 95* covers programming issues specific to the Windows 95 operating system. This guide provides conceptual and reference information that is not available in any other document.

The guide is divided into three main sections.

- "About Windows 95" discusses the Windows 95 architecture, Windows 95 system limits, and the API differences between version 3.x-based and Windows 95-based applications.
- "Using Windows 95 Features" discusses device I/O control, exclusive volume locking, long filenames, MS-DOS extensions, system policies, thunk compiler, ToolHelp functions, and virtual machine services.
- "Windows 95 Reference" contains complete reference information for the functions, structures, messages, and interfaces that are specific to Windows 95.

Windows 95 Architecture

The operating environment for Windows 95 consists of a computer's hardware devices and the following software components:

- Virtual machine manager (VMM).
- Virtual devices (VxDs).
- Device drivers.
- 16- and 32-bit Windows dynamic-link libraries (DLLs).
- MS-DOS - based applications.
- 16- and 32-bit Windows-based applications.

Virtual Machine Manager

The virtual machine manager (VMM) is the 32-bit protected-mode operating system at the core of Windows 95. Its primary responsibility is to create, run, monitor, and terminate virtual machines. The VMM provides services that manage memory, processes, interrupts, and exceptions such as general protection faults. The VMM works with virtual devices, 32-bit protected-mode modules, to allow the virtual devices to intercept interrupts and faults to control the access that an application has to hardware devices and installed software.

Both the VMM and virtual devices run in a single, 32-bit, flat model address space at privilege level 0 (also called ring 0). The system creates two global descriptor table (GDT) selectors, one for code and the other for data, and uses the selectors in the CS, DS, SS, and ES segment registers. Both selectors have a base address of zero and a limit of 4 gigabytes (GBs), so all the segment registers point to the same range of addresses. The VMM and virtual devices never change these registers.

The VMM provides multiple-threaded, preemptive multitasking. It runs multiple applications simultaneously by sharing CPU (central processing unit) time between the virtual machines in which the applications run. The VMM is also nonreentrant. This means that virtual devices must synchronize access to the VMM services. The VMM provides services, such as semaphores and events, to help virtual devices prevent reentering the VMM.

For more information about the VMM, including descriptions of the services that it provides to virtual devices, see the documentation included in the Microsoft Windows 95 Device Driver Kit (DDK).

Virtual Devices

Virtual devices (VxDs) are 32-bit programs that support the device-independent VMM by managing the computer's hardware devices and supporting software. VxDs support all hardware devices for a typical computer, including the programmable interrupt controller (PIC), timer, direct memory access (DMA) device, disk controller, serial ports, parallel ports, keyboard, and display adapter. A VxD is required for any hardware device that has settable operating modes or retains data over any period of time. In other words, if the state of the hardware device can be disrupted by switching between multiple virtual machines or applications, the device must have a corresponding VxD.

Some VxDs support software, but no corresponding hardware device. In general, a VxD can provide any kind of services for the VMM and other virtual devices. Windows 95 allows the user to install new virtual device drivers to support an add-on hardware device or provide some system-wide software service.

A VxD can also provide application programming interface (API) functions for applications running in virtual 8086 mode or protected mode. These functions can give applications direct access to the features of the VxD.

Windows 95 includes a device input and output control (IOCTL) interface that allows Microsoft® Win32® - based applications to communicate directly with VxDs. Applications typically use this interface to carry out selected MS-DOS system functions, to obtain information about a device, or to carry out input and output (I/O) operations that are not available through standard Win32 functions. For more information about the device IOCTL interface, see Device I/O Control.

For more information about virtual devices, see the documentation included in the Windows 95 DDK.

Device Drivers

A Windows device driver is a DLL that Windows uses to interact with a hardware device, such as a display or keyboard. Rather than access devices directly, Windows loads device drivers and calls functions in the drivers to carry out actions on the device. Each device driver exports a set of functions; Windows calls these functions to complete an action, such as drawing a circle or translating a keyboard scan code. The driver functions also contain the device-specific code needed to carry out actions on the device.

Windows requires device drivers for the display, keyboard, and communication ports. Other drivers may also be required if the user adds optional devices to the system.

The Windows 95 DDK provides independent hardware and software vendors (IHVs and ISVs) with the resources to build device drivers and VxDs that are compatible with the Windows 95 operating system. The resources include a configurable development environment, documentation, tools, and header files and libraries for several device types. The Windows 95 DDK contains the following components:

- Header files and libraries for building device drivers and VxDs.
- Sample source code for device drivers and VxDs.
- 16- and 32-bit versions of the driver development tools.

Dynamic-Link Libraries

Dynamic linking provides a mechanism for linking applications to libraries of functions at run time. The libraries reside in their own executable files and are not copied into an application's executable file as with static linking. These libraries are "dynamically linked" because they are linked to an application when it is loaded and executed rather than when it is linked. When an application uses a DLL, the operating system loads the DLL into memory, resolves references to functions in the DLL so that they can be called by the application, and unloads the DLL when it is no longer needed. Dynamic linking can be performed explicitly by applications or implicitly by the operating system.

DLLs are designed to provide resources to applications. Many applications can use the code in a DLL, meaning that only one copy of the code is resident in the system. Also, it is possible to update a DLL without changing applications that use the DLL, as long as the interface to the functions in the DLL does not change.

Software developers can extend the Windows environment by creating a DLL that contains routines for performing operations and then making the DLL available to other Windows-based applications (in addition to internal Windows routines). DLLs most often appear as files with a .DLL filename extension; however, they may also have an .EXE or other filename extension.

For more information about dynamic-link libraries, see Dynamic-Link Libraries.

Windows 95 supports 32-bit DLLs as well as 16-bit DLLs that were written for Windows version 3.*x*. For a discussion of the issues involved in mixing 16- and 32-bit components in the Windows 95 environment, see Thunk Compiler.

MS-DOS - Based Applications

Windows 95 supports applications written for MS-DOS. Each MS-DOS - based application can run as a full-screen application, or it can run in a window on the Windows 95 desktop.

The system can run multiple MS-DOS - based applications at the same time. To do so, it creates a separate virtual machine (VM) for each MS-DOS - based application and shares the microprocessor among the MS-DOS VMs and the system VM (which contains all Windows-based applications). A VM can run an MS-DOS - based application in either the virtual 8086 mode or protected mode of the microprocessor.

Although most MS-DOS - based applications run fine in a window or as a full-screen application, some may not. To ensure absolute backward compatibility for all MS-DOS - based applications, Windows 95 provides a separate operating mode called "single MS-DOS application mode." When in this mode, Windows 95 runs only one MS-DOS - based application at a time. No Windows-based applications run in that mode; in fact, none of the graphical user interface (GUI) components of the system are even loaded.

Windows 95 supports the complete set of MS-DOS system functions and interrupts and provides extensions that permit MS-DOS - based applications to take advantage of features such as long filenames, and exclusive volume locking. For more information, see MS-DOS Extensions, Long Filenames, and Exclusive Volume Locking.

Disk utilities and other applications that directly modify file system structures, such as directory entries, must request exclusive use of the volume before making modifications to the structures. Windows 95 provides a set of input and output control (IOCTL) functions to manage exclusive volume use. Exclusive use prevents applications from inadvertently changing the file system while a disk utility is trying modify it.

Virtual machine services let Microsoft MS-DOS - based applications take advantage of features provided by Windows 95 when the applications run in a window. MS-DOS - based applications can retrieve and, optionally, set the title of the window in which they run. Virtual machine services also allow MS-DOS - based applications to periodically check the state of an internal close flag and terminate if the flag is set. Windows 95 sets this flag when the user chooses the Close command from the system menu of the window in which the MS-DOS - based application runs. Close-aware applications enable the Close command, which gives the user an alternate way to exit the application and close the window. For more information, see Virtual Machine Services.

Windows-Based Applications

Windows 95 supports 16-bit applications written for Windows version 3.*x* as well as 32-bit applications that use the Win32 or Microsoft® Win32s® API. For 16-bit applications, Windows 95 preserves the cooperative multitasking model used in Windows version 3.*x*; that is, all 16-bit applications share the same virtual address space, the same message queue, and the same thread of execution. By contrast, each 32-bit Windows-based application has its own address space, a private message queue, and one or more threads of execution. In addition, each 32-bit thread is preemptively multitasked.

All new applications should be 32-bit applications developed using the Win32 API. For information about porting a 16-bit application to Win32, see Version Differences.

Windows 95 System Limitations

Microsoft® Windows® 95 implements some functions and messages differently than Microsoft® Windows NT®.

About Windows 95 System Limitations

To minimize development and debugging time when you target both Windows 95 and Windows NT, you need to understand the differences in the following areas:

- Windows 95 General Limitations
- Windows 95 Window Management
- Windows 95 Graphics Device Interface
- Windows 95 System Services
- Windows 95 Multimedia

Windows 95 General Limitations

Certain functions and classes of functions, such as for security and event logging, are not supported by Windows 95. Windows 95 provides stub routines for these unsupported functions so that applications designed for other operating systems that fully support the Win32 application programming interface (API) can run on Windows 95 without errors.

The extended error codes returned by the **GetLastError** function are not guaranteed to be the same in Windows 95 and Windows NT. This difference applies to extended error codes generated by calls to window management, GDI, and system services functions. For example, the **ActivateKeyboardLayout**, **GetKeyboardLayoutName**, and **UnloadKeyboardLayout** functions do not support extended error code values; that is, you cannot retrieve errors for these functions by using the **GetLastError** function.

By design, functions that take string parameters can handle either Unicode™ (wide character) or ANSI strings. However, Windows 95 does not implement the Unicode (or wide character) version of most functions. With few exceptions, these functions are implemented as stubs that simply return an error value. However, Windows 95 does provide Unicode implementations of the following functions.

ExtTextOut
GetCharWidth
GetTextExtentExPoint
GetTextExtentPoint
MessageBox
MessageBoxEx
TextOut

In addition, Windows 95 implements the **MultiByteToWideChar** and **WideCharToMultiByte** functions for converting strings to and from Unicode.

Windows 95 Window Management

Windows 95 implements some window management features in 16 bits. The use of 16 bits imposes some restrictions on parameters in functions and messages and places limits on internal storage. In some cases, Windows 95 provides features that can be used to avoid these restrictions and limitations. For example, the standard edit control is limited to somewhat less than 64 kilobytes (K) of text; however, the text in a rich edit control is limited only by available memory.

Handles

Windows 95 permits up to 16,364 window handles and 16,364 menu handles. Although Windows NT supports more handles than this, Windows 95 does support significantly more handles than Windows version 3.1.

Messages

The *wParam* parameter for the **SendMessageCallback**, **SendMessageTimeout**, and **SendNotifyMessage** functions is limited to a 16-bit value.

In Windows 95, the *wParam* parameter in list box messages, such as LB_INSERTSTRING or LB_SETITEMDATA, is limited to a 16-bit value. One effect of this limit is that list boxes cannot contain more than 32,767 items. Although the number of items is restricted, the total size, in bytes, of the items in a list box is limited only by available memory. In contrast, a 64K data limit is imposed by Windows version 3.1.

Any private application message must be defined above WM_USER + 0x100. A value above this will ensure that there is no collision between private messages and dialog box control messages.

Three-Dimensional Look

Windows 95 automatically applies the standard three-dimensional shading and color scheme to dialog boxes created by applications marked as version 4.0 or later. Applications that are marked for earlier versions can still get the three-dimensional appearance by applying the DS_3DLOOK style to dialog boxes. If this style is used, the system automatically applies the three-dimensional look without requiring the application to check the operating system version. This is useful, for example, in applications developed for Windows NT version 3.5. The DS_3DLOOK style is ignored in Windows NT version 3.1.

Message-box Styles

In Windows 95, the MB_ICONQUESTION style used with the **MessageBox**, **MessageBoxEx**, and **MessageBoxIndirect** functions is obsolete. Win32-based applications should use the MB_ICONEXCLAMATION style instead. Similarly, applications should use the MB_ICONINFORMATION style instead of the MB_ICONASTERISK style and the MB_ICONSTOP style instead of the MB_ICONHAND style.

Cursors

The IDC_SIZE and IDC_ICON values used with the **LoadCursor** function are obsolete and should not be used in a Windows 95 - based application or in a Win32-based application that is marked as version 4.0.

Desktop

In Windows 95, only one desktop is available while the system runs. Although the thread desktop functions, **GetThreadDesktop** and **SetThreadDesktop**, are available under Windows 95, they do not do anything.

Windows 95 Graphics Device Interface

Windows 95 implements some GDI features in 16 bits. The use of 16 bits imposes some restrictions on parameters in functions and places limits on internal storage.

Coordinate System

Windows 95 uses a 16-bit world coordinate system and restricts x- and y- coordinates for text and graphics to the range ±32K. Windows NT uses a 32-bit world coordinate system and allows coordinates in the range ±2 gigabytes (GB). If you pass full 32-bit coordinates to text and graphics functions in Windows 95, the system truncates the upper 16 bits of the coordinates before carrying out the requested operation.

Because Windows 95 uses a 16-bit coordinate system, the sum of the coordinates of the bounding rectangle specified by the **Arc**, **Chord**, **Pie**, **Ellipse**, and **RoundRect** functions cannot exceed 32K. In addition, the sum of the *nLeftRect* and *nRightRect* parameters or the *nTopRect* and *nBottomRect* parameters cannot exceed 32K.

World Transformations

Windows 95 does not support world transformations that involve either shearing or rotations. The **ExtCreateRegion** function fails if the transformation matrix is anything other than a scaling or translation of the region.

Bitmaps

Windows 95 does not support the CBM_CREATEDIB value for the **CreateDIBitmap** function. The **CreateDIBSection** function should be used instead to create a DIB. **CreateDIBSection** is also available in Windows NT version 3.5.

If the **biCompression** member of the **BITMAPINFOHEADER** structure is the BI_BITFIELDS value, the **bmiColors** member of the **BITMAPINFO** structure contains three doubleword color masks that specify the red, green, and blue components, respectively, of each pixel. Windows 95 only supports the following color masks for 16 and 32 bits per pixel (bpp).

16bpp	The blue mask is 0x001F, the green mask is 0x03E0, and the red mask is 0x7C00.
16bpp	The blue mask is 0x001F, the green mask is 0x07E0, and the red mask is 0xF800.
32bpp	The blue mask is 0x000000FF, the green mask is 0x0000FF00, and the red mask is 0x00FF0000.

If the *lpvBits* parameter is NULL, the **GetDIBits** function fills in the dimensions and format of the bitmap in the **BITMAPINFO** structure pointed to by the *lpbi* parameter. In this case, if the function is successful, the return value in Windows 95 is the total number of scan lines in the bitmap. In Windows NT versions 3.1 and 3.5, however, the return value is 1 (TRUE), indicating success.

Pens and Brushes

In Windows 95, pens and brushes have several limitations. The **ExtCreatePen** function supports solid colors only (the PS_SOLID style), and the PS_ALTERNATE and PS_USERSTYLE styles are not supported. Geometric pens (the PS_GEOMETRIC style) are limited to the BS_SOLID brush style specified in the **LOGBRUSH** structure passed to **ExtCreatePen.** In addition, the following pen styles are supported in paths only.

PS_ENDCAP_FLAT	PS_JOIN_BEVEL
PS_ENDCAP_ROUND	PS_JOIN_MITER
PS_ENDCAP_SQUARE	PS_JOIN_ROUND

Windows 95 does not support the dashed or dotted pen styles, such as PS_DASH or PS_DOT, in wide lines. The BS_DIBPATTERN brush style is limited to an 8- by 8-pixel brush.

Windows 95 does not support brushes from bitmaps or device independent bitmaps (DIBs) that are larger than 8 by 8 pixels. Although bitmaps larger than 8 by 8 pixels can be passed to the **CreatePatternBrush** or **CreateDIBPatternBrush** function, only a portion of the bitmap is used to create the brush.

Windows 95 does not provide automatic tracking of the brush origin. An application is responsible for using the **UnrealizeObject**, **SetBrushOrgEx**, and **SelectObject** functions each time it paints using a pattern brush.

Logical Objects

In Windows 95, regions are allocated from the 32-bit heap and can, therefore, be as large as available memory. All other logical objects, however, share the 64K local heap. In addition, the number of region handles cannot exceed 16K.

To ensure that adequate space is always available for logical objects, applications should always delete objects when no longer needed. The following functions create objects that are placed in the local heap and have corresponding functions used to delete the objects.

Object	Create with	Delete with
Bitmap	**CreateBitmap**, **CreateBitmapIndirect**, **CreateCompatibleBitmap**, **CreateDIBitmap**, **CreateDIBSection**, **CreateDiscardableBitmap**	**DeleteObject**
Brush	**CreateBrushIndirect**, **CreateDIBPatternBrush**, **CreateDIBPatternBrushPt**, **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**	**DeleteObject**
Color space	**CreateColorSpace**	**DeleteColorSpace**
Device context (DC)	**CreateDC**, **GetDC**	**DeleteDC**, **ReleaseDC**
Enhanced metafile	**CloseEnhMetaFile**, **CopyEnhMetaFile**, **GetEnhMetaFile**, **SetEnhMetaFileBits**	**DeleteEnhMetaFile**
Enhanced metafile DC	**CreateEnhMetaFile**	**CloseEnhMetaFile**
Extended pen	**ExtCreatePen**	**DeleteObject**
Font	**CreateFont**, **CreateFontIndirect**	**DeleteObject**
Memory DC	**CreateCompatibleDC**	**DeleteDC**
Metafile	**CloseMetaFile**, **CopyMetaFile**, **GetMetaFile**, **SetMetaFileBitsEx**	**DeleteMetaFile**
Metafile DC	**CreateMetafile**	**CloseMetaFile**
Palette	**CreatePalette**	**DeleteObject**

Pen	**CreatePen**, **CreatePenIndirect**	**DeleteObject**
Region	**CombineRgn**, **CreateEllipticRgn**, **CreateEllipticRgnIndirect**, **CreatePolygonRgn**, **CreatePolyPolygonRgn**, **CreateRectRgn**, **CreateRectRgnIndirect**, **CreateRoundRectRgn**, **ExtCreateRegion**, **PathToRegion**	**DeleteObject**

Physical objects have always existed in global memory and are, therefore, not limited.

Object Deletion

Deletion of drawing objects is slightly different in Windows 95 than in Windows NT. In Windows NT, if a drawing object (pen or brush) is deleted while it is still selected into a DC, the **DeleteObject** function fails. In Windows 95, the function succeeds, but the result is a nonfunctioning object. This nonfunctioning object is automatically destroyed when the DC is deleted.

Paths

When a path is constructed in Windows 95, only the following functions are recorded: **ExtTextOut**, **LineTo**, **MoveToEx**, **PolyBezier**, **PolyBezierTo**, **Polygon**, **Polyline**, **PolylineTo**, **PolyPolygon**, **PolyPolyline**, and **TextOut**.

Graphics Mode and Device Capabilities

In Windows 95, the **GetGraphicsMode** and **SetGraphicsMode** functions support the GM_COMPATIBLE value only. The GM_ADVANCED value is not supported.

In Windows 95, the **DeviceCapabilities** function returns - 1 when called with the DC_FILEDEPENDENCIES value because that capability is not supported. In Windows 95, **DeviceCapabilities** supports the following additional capabilities.

DC_DATATYPE_PRODUCED	Retrieves an array of strings containing the data types that the printer driver supports. A return value of - 1 indicates that the printer driver only understands device-specific commands (in other words, "RAW" data) that are native to the printer. A return value of 2 or more indicates the number of strings in the array.
DC_EMF_COMPLIANT	Returns a flag that indicates if the specified printer driver is capable of accepting an enhanced metafile (EMF) spooled by the system (that is, the printer driver is EMF-compliant). The function returns 1 if the printer driver is EMF-compliant and - 1 if the printer driver is not.

Enhanced Metafiles

Although Windows 95 imposes no restrictions on the **PlayEnhMetaFile** and **PlayEnhMetaFileRecord** functions, the files and records that these functions execute are subject to the limitations described in this section. For example, the functions ignore records that attempt to draw outside of the 16-bit coordinate

space or that apply shearing or rotation to world transformations.

In Windows 95, the maximum length of the description string for an enhanced metafile is 16K. This limit applies to the **GetEnhMetaFileDescription**, **GetEnhMetaFileHeader**, and **GetEnhMetaFile** functions.

In Windows 95, the **dmDeviceName** member of the **DEVMODE** structure specifies the "friendly" name of the printer, which may be set to any user-defined value. Windows 95, however, does not support the the following members; they are included for compatibility with Windows NT.

dmBitsPerPel	**dmFormName**
dmDisplayFlags	**dmPelsHeight**
dmDisplayFrequency	**dmPelsWidth**

Printing and Print Spooler

Windows 95 does not support print monitor dynamic-link libraries (DLLs) that have been developed for Windows NT. To add a monitor using the **AddMonitor** function, you must specify a monitor DLL that has been explicitly created for Windows 95. The following printing and print spooling functions are not available in Windows 95.

AddForm	**FindFirstPrinterChangeNotification**
AddPrinterConnection	**FindNextPrinterChangeNotification**
ConnectToPrinterDlg	**GetForm**
DeleteForm	**ResetPrinter**
DeletePrinterConnection	**SetForm**
EnumForms	**WaitForPrinterChange**
FindClosePrinterChangeNotification	

In Windows 95, the **SetPrinter** function ignores the **pShareName** member of the **PRINTER_INFO_2** structure. The **PRINTER_INFO_3** and **PRINTER_INFO_4** structures used with the **SetPrinter**, **GetPrinter**, and **EnumPrinters** functions are not supported in Windows 95. The **PRINTER_INFO_5** structure, which is available in Windows 95, is not supported in Windows NT versions 3.1 and 3.5.

The PRINTER_ENUM_CONNECTIONS value used with the **EnumPrinters** function is not supported in Windows 95. The **DOC_INFO_2** structure used with the **StartDocPrinter** function and the **PORT_INFO_2** structure used with the **EnumPorts** function are not supported in Windows NT versions 3.1 and 3.5.

Windows 95 supports the DRAWPATTERNRECT printer escape.

Win32-based applications that send output to PostScript™ printers should use the **GetDeviceCaps** function to check for the PC_PATHS value to determine whether to use path functions or printer escapes to draw paths. Applications should use paths functions whenever possible. The following example shows how to check for this capability.

```
// Determine whether to use path functions on the device.
// hDC is the output device.

OSVERSIONINFO osvi;

osvi.dwOSVersionInfoSize = sizeof(osvi);
GetVersionEx(&osvi);

if ((osvi.dwPlatformId == VER_PLATFORM_WIN32_NT) ||
     ((osvi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS) &&
       (GetDeviceCaps(hDC, POLYCAPS) & PC_PATHS)))
    bUsePaths = TRUE;
else
    bUsePaths = FALSE;
```

In Windows 95, printer drivers typically set this capability to zero. This means that Win32-based applications sending output to Postscript printers need to use the **ExtEscape** function and the printer-specific escapes to draw paths at the printer. (The **Escape** function cannot be used for this.)

In Windows NT, the string specified in the *lpszDeviceName* parameter of the **EnumDisplaySettings** function must be of the form "\\.\Display*X*", where *X* can be 1, 2, or 3. In Windows 95, *lpszDeviceName* must be NULL.

Windows 95 System Services

File I/O

Windows 95 does not support asynchronous file input and output (I/O), except on serial devices. Therefore, the **ReadFile** and **WriteFile** functions will fail if you pass in an overlapped region on anything other than a serial device. The **GetOverlappedResult** function works only on serial devices or on files opened by using the **DeviceIoControl** function.

In Windows 95, the **ReadFileEx** and **WriteFileEx** functions will fail if you pass in the handle of a serial device (for example, COM2). **ReadFile** and **WriteFile**, however, accept the handle of a serial device.

In Windows 95, the **CreateFile** function does not support the standard "\\.\C:" and "\\.\PhysicalDrive0" formats used to gain access to the logical or physical drives. To gain access, applications must specify a virtual device (VxD) name instead and use the **DeviceIoControl** function to send requests through the VxD to the logical and physical drives. For more information, see Device I/O Control.

In Windows NT, the **DeleteFile** function fails if you attempt to delete a file that is open for normal I/O or is opened as a memory mapped file. In Windows 95, **DeleteFile** deletes such files. Because deleting open files may cause loss of data and application failure, you must take every precaution to close files before attempting to delete them by using **DeleteFile**.

Time

In Windows NT, the **FileTimeToDosDateTime** and **DosDateTimeToFileTime** functions allow dates up to 12/31/2107. In Windows 95, these functions allow dates up to 12/31/2099.

The precision of the time for a file on a file allocation table (FAT) file system volume is 2 seconds. If Windows 95 is connected through a network to a different file system, the time precision is limited only by the remote device.

Memory Management

In Windows 95, fixed memory blocks cannot be reallocated to be movable. The GMEM_MODIFY and GMEM_MOVEABLE combination of values has no effect when a memory block is reallocated by using the **GlobalReAlloc** function. Similarly, the LMEM_MODIFY and LMEM_MOVEABLE combination has no effect when a memory block is reallocated by using the **LocalReAlloc** function.

In Windows 95, committing memory for a page that is already committed is an expensive operation that has no ultimate effect (it is expensive because additional storage is allocated and subsequently freed for each committed page). When committing memory by using the **VirtualAlloc** function, an application should specify only the pages that actually need to be committed.

Although applications can request that memory allocation be at a specific virtual address, applications must not depend on any given address range always being available on every operating system. Applications can query the address space by using the **GetSystemInfo** function.

In Windows 95, memory allocated by Win32-based applications falls in the address range 4 megabytes (MB) - 2GB for private memory and 2GB - 3GB for shared memory (shared mapped files). The PAGE_WRITECOPY and PAGE_GUARD access protection values are not supported. Instead of using the PAGE_GUARD value and handling the EXCEPTION_GUARD_PAGE exception, applications can use the PAGE_NOACCESS value and handle the EXCEPTION_ACCESS_VIOLATION exception.

File Mapping

The SEC_IMAGE and SEC_NOCACHE values for the *fdwProtect* parameter of the **CreateFileMapping** function are not supported in Windows 95. In addition, the *dwMaximumSizeHigh* parameter of **CreateFileMapping** is ignored in Windows 95, so applications should specify zero for the parameter.

In Windows 95, shared memory mapped files that are created by using the **MapViewOfFileEx** function appear in the same address space across all 32-bit processes in the system. If you pass in a specific base offset in the *lpvBase* parameter of **MapViewOfFileEx** and the function succeeds, you are guaranteed that the same memory region is available in every process. This is not true in Windows NT because **MapViewOfFileEx** fails for any process that already has the given memory region in use.

Coherence guarantees that the data accessible in a file view is an identical copy of the file's contents on disk. In Windows 95, file views derived from a single file-mapping object are coherent only if the file is accessed through one of the views. A view of a file is not guaranteed to be coherent if the file is accessed by normal file I/O functions, such as **ReadFile** or **WriteFile**, or by views created from a different file-mapping object.

If you close a file handle that was used to create a file mapping object, both Windows NT and Windows 95 hold the file open until you unmap the last view of the file by using the **UnmapViewOfFile** function. However, Windows NT holds the file open with no sharing restrictions, whereas Windows 95 holds it open using the sharing restrictions of the original file handle. To ensure exclusive access to a file in Windows NT, the file handle must remain open for the life of the file-mapping object. Because Windows 95 retains the sharing restrictions, both the file handle and the handle to the file-mapping object may be closed after calling **MapViewOfFile** and exclusive access to the file is ensured.

In Windows 95, if the FILE_MAP_COPY value is specified for the *fdwAccess* parameter of **MapViewOfFile** (or **MapViewOfFileEx**), the *hMapObject* parameter must have been created with the PAGE_WRITECOPY value. In addition, the *dwOffsetHigh* parameter of **MapViewOfFile** (or **MapViewOfFileEx**) is ignored, so applications should specify zero for the parameter.

Windows 95 implements copy-on-write file mappings slightly differently than Windows NT. In Windows 95, a call to **MapViewOfFile** with FILE_MAP_COPY returns an error unless PAGE_WRITECOPY was used with the **CreateFileMapping** function. In both Windows NT and Windows 95, creating the map with PAGE_WRITECOPY and the view with FILE_MAP_COPY produces a view to the file that makes the pages swappable and prevents modifications from going to the original data file. In Windows 95, PAGE_WRITECOPY must be passed to **CreateFileMapping**, but this is optional in Windows NT.

If you share the mapping between multiple processes by using the **DuplicateHandle** or **OpenFileMapping** function and one process writes to a view, the modifications will not be propagated to the other process in Windows NT. However, the modifications will be propagated in Windows 95. The original file, though, will not change on either platform.

Registry

The Windows 95 registry does not allow key names containing control characters. In addition, if the *lpszSubKey* parameter is an empty string (""), the **RegDeleteKey** function deletes the key identified by the *hKey* parameter.

In Windows 95, the **RegCreateKeyEx** function creates a non-volatile key even if the REG_OPTION_VOLATILE value is specified.

Handle Duplication

In Windows 95, the **DuplicateHandle** function cannot duplicate handles of registry keys as it can in Windows NT. The function returns an error if an application attempts to duplicate the handle of a registry key. In addition, when a file handle is duplicated, the duplicated handle will not be granted more access than the original.

Thread Locales

Thread locales, retrieved and set by using the **GetThreadLocale** and **SetThreadLocale** functions, are static and can only be changed at system boot time.

Instruction Cache

In Windows 95, the **FlushInstructionCache** function always returns TRUE. Windows 95 supports single processor machines only.

Resources

In Windows 95, a call to the **FreeResource** function must be included for every call to the **LoadResource** function. The call to **FreeResource** allows the system to discard a resource that an application no longer needs. Windows NT automatically frees resources, so a call to **FreeResource** is not required.

Access Rights

The SYNCHRONIZE standard access rights flag is not supported in Windows 95. The following functions are affected.

DuplicateHandle
MsgWaitForMultipleObjects
OpenEvent
OpenMutex
OpenProcess
OpenSemaphore
WaitForMultipleObjects
WaitForMultipleObjectsEx
WaitForSingleObject
WaitForSingleObjectEx

Windows 95 Multimedia

The **sndAlias** macro is not supported in Windows 95. In addition, the SND_ALIAS and SND_ALIAS_ID values for the **PlaySound** function are not supported in Windows 95.

The Windows 95 multimedia functions are not designed to be used by two or more threads in the same process. Although most multimedia functions will work if they are called by multiple threads, some are likely to fail. Functions that are particularly likely to fail include **PlaySound**, any of the functions that prepare or unprepare headers, and any of the open and close functions. **PlaySound** can never be used simultaneously by multiple threads in the same process. The functions that prepare or unprepare headers and the open and close functions can be used simultaneously by multiple threads in the same process, but only if they do not pass the same structure.

Version Differences

The Microsoft® Windows® 95 operating system supports Windows-based applications that have a subsystem version number of either 3.*x* or 4.0. An application's subsystem version number is set by the linker.

This overview describes the differences in the way Windows 95 treats applications based on their subsystem version numbers. It is intended to help you identify areas in an application written for Windows version 3.*x* that you must revise to take advantage of the new features provided by Windows 95. It is divided into the following sections:

Window Management Differences

When a version 4.0 application uses the **SetWindowLong** function (with GWL_STYLE) to change a window's style, Windows 95 sends the window a WM_STYLECHANGING message before changing the style. The message's *lParam* parameter is the address of a **STYLESTRUCT** structure. The **styleOld** and **styleNew** members of the structure specify the old and new styles. By processing WM_STYLECHANGING, an application can inspect the styles and perhaps change them.

Windows 95 sends the WM_STYLECHANGED message after changing the style. Again, the *lParam* parameter is the address of a **STYLESTRUCT** structure that specifies the new styles. The application can use WM_STYLECHANGED to update any style-dependent information stored in the application's internal data structures.

Windows 95, however, does not send the WM_STYLECHANGING and WM_STYLECHANGED messages to a version 3.*x* application.

A version 4.0 application cannot use the **SetWindowLong** function to set the WS_EX_TOPMOST style for a window or to remove the style from a window. The application must use the **SetWindowPos** function to set or remove the WS_EX_TOPMOST style.

Windows 95 automatically adds and removes the WS_EX_WINDOWEDGE style for windows in both version 3.*x* and 4.0 applications. In a version 3.*x* application, Windows 95 adds the WS_EX_WINDOWEDGE style to a window if, in version 3.1, the window would have a dialog border or a sizable border. Windows 95 removes the WS_EX_WINDOWEDGE style if the window's style changes so that it would no longer have a dialog border or sizable border in version 3.1. Windows 95 uses similar criteria for adding and removing the WS_EX_WINDOWEDGE style for a Windows version 4.0 application, except that any window that has a title bar receives the WS_EX_WINDOWEDGE style, regardless of the window's other border styles.

When the user drags the icon of a minimized window created by a version 3.*x* application, Windows 95 sends the window a WM_QUERYDRAGICON message to retrieve the cursor to use while dragging. Windows 95 also sends WM_QUERYDRAGICON to retrieve the icon to display in the task-switch window that appears when the user presses the ALT+TAB key combination. Windows 95 does not send WM_QUERYDRAGICON to a window created by a version 4.0 application. Instead, the application is expected either to use the WM_SETICON and WM_GETICON messages or to set the big and small icons when registering the window class.

When a window in a version 4.0 application loses the mouse capture as a result of a call to the **SetCapture** function, the window receives a WM_CAPTURECHANGED message, but Windows 95 sends the message asynchronously. In other words, the window receives the message, but possibly not right away. Some of the ways in which a window can lose the mouse capture include:

- The user activated a different application by clicking one of its windows.
- The **DefWindowProc** function changed the capture in response to a WM_CANCELMODE message.
- Another window using the same message queue called the **SetCapture** function. (All 16-bit applications share the same queue, but each 32-bit thread has its own queue.)

If a child window in a version 3.*x* application has the WS_EX_NOPARENTNOTIFY style, Windows 95 disregards the style when the user clicks the child window. That is, Windows 95 sends the WM_PARENTNOTIFY message to all windows in the parent chain regardless of whether the child window has the WS_EX_NOPARENTNOTIFY style. If a child window in a version 4.0 application has this style, Windows 95 does not send WM_PARENTNOTIFY messages when the user clicks the child window.

In a version 3.*x* application, it is possible for the horizontal coordinate on the left side of a window's client area to be greater than that on the right side. This happens because version 3.*x* sometimes incorrectly

handles an empty client rectangle that contains a vertical scroll bar. (Fixing the problem would cause some applications to generate general protection faults.) In a version 4.0 application, it is not possible for the horizontal coordinate of the left side of a client area to be greater than that of the right side.

Dialog Box Differences

A dialog box created by a version 4.0 application automatically receives the DS_3DLOOK style. This style gives three-dimensional borders to child controls in the dialog box and draws the entire dialog box using the three-dimensional color scheme. The DS_3DLOOK style is available to a dialog box created by a version 3.*x* application, but you must explicitly add the style to the dialog box template. The application that creates the dialog box determines the version number of the dialog box.

Windows 95 performs a strict validation check on the DS_ styles specified in a dialog box template. If the template contains any styles that Windows 95 does not recognize and a version 4.0 application is creating the dialog box, the creation fails. If a version 3.*x* application is creating the dialog box, the system debugger generates a warning, but Windows 95 creates the dialog box anyway.

Button Differences

The parent window of a button (except push buttons) in a version 3.*x* application receives a WM_CTLCOLORBTN message when the button is about to be drawn. In a version 4.0 application, however, the parent window of a button receives the WM_CTLCOLORSTATIC message, which retrieves a color appropriate for drawing text on the background of the dialog box. Windows 95 sends WM_CTLCOLORSTATIC to retrieve the background and text colors for the text area of check boxes, radio buttons, and group buttons. An application should process WM_CTLCOLORSTATIC in order to correctly set the colors of any dialog box item that contains text and appears directly on the dialog area.

Windows 95 performs default handling of the WM_CTLCOLORBTN message differently depending on an application's version. For a version 3.*x* application, the default handling for button colors is to use the COLOR_WINDOW value for the background color and the COLOR_WINDOWTEXT value for the foreground color. For a version 4.0 application, Windows 95 uses the COLOR_3DFACE value for the background and the COLOR_BTNTEXT value for the foreground.

In a version 3.*x* application, a push button's outer top left corner is nonwhite because the button is typically drawn on a white background. If the border was white, the background would appear to bleed into the button. In a version 4.0 application, a push button's outer top left corner is white (COLOR_3DHILIGHT) because the button is typically drawn on a nonwhite background (COLOR_3DFACE).

Edit Control Differences

In a version 3.*x* application, an edit control that is the descendant of an inactive window takes the input focus when the user clicks the control; an edit control in a version 4.0 application does not. Not taking the input focus prevents the situation where the user can enter text into what appears to be an inactive window.

An edit control in a version 3.*x* application retrieves its text and background colors by sending the WM_CTLCOLOREDIT message to its parent window. In a version 4.0 application, an edit control sends the WM_CTLCOLOREDIT or WM_CTLCOLORSTATIC message. The edit control sends WM_CTLCOLORSTATIC if it is disabled or read-only; otherwise, it sends WM_CTLCOLOREDIT. In addition, a disabled multiline edit control in a version 4.0 application uses the COLOR_GRAYTEXT value as its text color.

A multiline edit control in a version 4.0 application has a proportional scroll box (thumb), but a multiline edit control in a version 3.*x* application does not.

In a version 3.*x* application, the *wParam* parameter of the EM_REPLACESEL message is not used. In a version 4.0 application, the *wParam* parameter is a flag that specifies whether the replacement operation can be undone.

List Box Differences

In a version 4.0 application, a list box that is part of a combo box uses the WM_CAPTURECHANGED notification message to hide its drop-down list if it is open. For more information, see "Combo Boxes" later in this topic.

The DDL_EXCLUSIVE flag of the **DlgDirList** function does not have the expected result in a version 3.*x* application. Specifically, the flag does not exclude read-write files from the list. In a version 4.0 application, the DDL_EXCLUSIVE flag excludes read-write files.

If a list box in a version 3.*x* application has either the WS_HSCROLL or WS_VSCROLL style, the list box receives both horizontal and vertical scroll bars. Although one of the scroll bars is typically hidden, Windows 95 displays the hidden scroll bar if its scrolling range becomes greater than zero. In a version 4.0 application, a list box does not receive a horizontal scroll bar, unless it has the WS_HSCROLL style. Likewise, it does not receive a vertical scroll bar unless it has the WS_VSCROLL style.

When creating a list box in a version 3.*x* application, Windows 95 always increases the size of the list box by adding the border width to each side. This is done because Windows 95 assumes that the dimensions specified by the application or the dialog template are for the client area of the list box. Unfortunately, increasing the size in this way makes aligning a list box rather difficult. Windows 95 does not increase the size when creating a list box in a version 4.0 application; Windows 95 assumes that the specified size includes the borders.

Combo Box Differences

A combo box in a version 4.0 application passes the control color messages (WM_CTLCOLOR*) from its child components (edit control and list box) to the parent window of the combo box. In a version 3.*x* application, a combo box passes those messages to the **DefWindowProc** function.

A combo box in a version 4.0 application uses the WM_CAPTURECHANGED message to hide its drop-down list box if it is open. Windows 95 sends the message when another window takes the mouse capture, which typically happens when the user clicks another window. In a version 3.*x* application, a combo box does not use WM_CAPTURECHANGED to hide the drop-down list.

A combo box in a version 3.*x* application uses the WM_CTLCOLORLISTBOX message to retrieve the text and background colors. In a version 4.0 application, a combo box uses the WM_CTLCOLOREDIT or WM_CTLCOLORSTATIC message instead. The combo box uses WM_CTLCOLORSTATIC if it is disabled or contains a read-only selection field (in an edit control); otherwise, it uses WM_CTLCOLOREDIT.

In a version 3.*x* application, the background of the static text area in read-only combo boxes is filled with the system highlight color (COLOR_HILIGHT). In a version 4.0 application, Windows 95 fills the background of the static text area only for a combo box that is not owner drawn.

In a version 4.0 application, Windows 95 adds the ODS_COMBOBOXEDIT value to the **itemState** member of the **DRAWITEMSTRUCT** structure when Windows 95 sends the WM_DRAWITEM message to the parent window of an owner-drawn combo box to draw an item in the selection field. The ODS_COMBOBOXEDIT value tells the parent window that the drawing takes place in the selection field of the combo box rather than in the list box.

Menu Differences

A 16-bit version 3.*x* application that creates or loads a menu is considered to be the owner of the menu. However, when the application exits, the menu is "orphaned" until no 16-bit version components for Windows version 3.*x* remain. For 16-bit version 4.0 applications and all 32-bit applications, the application that creates the menu is the owner, and the menu is destroyed as soon as the application exits. Unlike graphics device interface (GDI) objects, there is no way to change the ownership of a menu.

In a version 3.*x* application, the Close command cannot be deleted from the System menu of an multiple document interface (MDI) child window. In a version 4.0 application, the Close command can be deleted.

Windows 95 increases the width of hierarchical, owner-drawn menu items in a version 3.*x* application. Some applications rely on this increased width and use it to include icons that simulate toolbars. In a version 4.0 application, Windows 95 does not automatically increase the width of hierarchical, owner-drawn menu items.

The *wParam* parameter of the WM_MENUSELECT message is interpreted differently depending on the subsystem version number of the application and whether the application is written for 16 or 32 bits:

- In a 16-bit version 3.*x* application, the *wParam* parameter is the handle of the pop-up menu if the selected item activates a pop-up menu.
- In a 16-bit version 4.0 application, *wParam* is the identifier of the menu item, regardless of whether the item activates a pop-up menu.
- In a 32-bit application (both subsystem versions), the low-order word of *wParam* is the identifier of the menu item or, if the item activates a pop-up menu, the index of the pop-up menu. This high-order word contains the menu flags.

System Bitmap and Color Differences

In 16-bit version 3.*x* applications, purpose windows, such as dialog boxes, message boxes, and system-defined control windows, receive WM_CTLCOLOR messages when they are about to be drawn. The high-order word of the *lParam* parameter indicates the type of special purpose window about to be drawn (CTLCOLOR_BTN, CTLCOLOR_EDIT, and so on). By default, a special purpose window passes the WM_CTLCOLOR message to the parent or owner window (for both subsystem versions), allowing the parent or owner to set the foreground and background colors of the special purpose window. The same is true for 32-bit applications, except that the special purpose window receives one or more of the following messages instead of WM_CTLCOLOR.

WM_CTLCOLORBTN	WM_CTLCOLORMSGBOX
WM_CTLCOLORDLG	WM_CTLCOLORSCROLLBAR
WM_CTLCOLOREDIT	WM_CTLCOLORSTATIC
WM_CTLCOLORLISTBOX	

In 32-bit applications, the message itself indicates the type of special purpose window about to be drawn. The *lParam* parameter contains the window's 32-bit handle.

Display drivers for Windows version 3.*x* provide the bitmaps, icons, and cursors used by previous versions of Windows. Because Windows 95 renders and scales the system bitmaps, icons, and cursors itself, its display drivers do not (and should not) contain any OBM_, OIC_, or OCR_ resources.

For a version 3.*x* application, the default handling of the WM_CTLCOLORSTATIC and WM_CTLCOLORDLG messages is to use the COLOR_WINDOW value for the background and the COLOR_WINDOWTEXT value for the foreground. For a version 4.0 application, Windows 95 uses the COLOR_3DFACE value for the background and the COLOR_WINDOWTEXT value for the foreground.

When an application calls the **GetClientRect** function to retrieve the client rectangle of a minimized window created by a version 3.*x* application, Windows 95 retrieves the old dimensions for a minimized window (0, 0, 36, 36). For a minimized window in a version 3.1 or 4.0 application, Windows 95 retrieves 0, 0, **GetSystemMetrics**(SM_CXMINIMIZED), **GetSystemMetrics**(SM_CYMINIMIZED). These metrics change if the user changes the title bar height or minimized window width by using Control Panel. In other words, the **GetClientRect** function returns the dimensions of the entire minimized window, preventing an application from causing a general protection (GP) fault because of an unexpectedly empty client rectangle.

System Metrics Differences

When a version 3.*x* application calls the **GetSystemMetrics** function to retrieve the SM_CYVSCROLL or SM_CYHSCROLL metric value, the function returns a value that is one pixel more than the actual height of the corresponding type of standard scroll bar. Windows 95 adds a pixel because applications written for previous versions of windows routinely subtract one pixel from the return value. Subtracting one pixel accounts for the way a standard scroll bar in a version 3.*x* application overlaps the border of the window in which it resides. A version 4.0 application receives the actual height of the scroll bar.

When a version 3.*x* application retrieves the SM_CXDLGFRAME and SM_CYDLGFRAME system metric values, **GetSystemMetrics** returns a value that is one pixel less that the actual frame width or height. A version 4.0 application receives the actual width or height.

GetSystemMetrics returns one pixel more than the actual height of a title bar when a version 3.*x* application requests the SM_CYCAPTION system metric value. A version 4.0 application receives the actual height of the title bar.

GetSystemMetrics returns one pixel less than the actual height of a menu bar when a version 3.*x* application requests the SM_CYMENU system metric value. A version 4.0 application receives the actual height of the menu bar.

When a version 3.*x* application calls **GetSystemMetrics** to retrieve the SM_CYFULLSCREEN value (height of a maximized window's client area), the function returns a value that is one pixel less than the actual height. This is because **GetSystemMetrics** returns one pixel more than the actual title bar height when an application retrieves the SM_CYCAPTION value. (The sum of the height of a maximized window's client area and the height of a title bar must equal the height of the working area of the screen.) A version 4.0 application receives the actual height of the maximized window's client area when it requests the SM_CYFULLSCREEN value.

Parameter Validation Differences

If a 32-bit version 3.*x* application specifies invalid class styles when calling the **RegisterClass** function, Windows 95 strips out the invalid bits and generates warnings in the system debugger, but allows **RegisterClass** to succeed anyway. If a 32-bit version 4.0 application passes invalid class styles to **RegisterClass**, the function fails.

In a version 3.*x* application, Windows 95 does not validate the **length** member of the **WINDOWPLACEMENT** structure that is passed to the **GetWindowPlacement** and **SetWindowPlacement** functions. The **length** member, however, is validated for a version 4.0 application; Windows 95 fails these functions if the value of **length** is incorrect.

Windows 95 does not validate the **cbSize** member of the **STARTUPINFO** structure specified in the **CreateProcess** and **GetStartupInfo** functions for applications written for Windows version 3.*x*. The **cbSize** member is validated, however, for version 4.0 applications.

In the debugging version of Windows 95, the system fills the specified buffer with zeros up to the length specified by the *cbSize* parameter when a version 4.0 application calls the **LoadString** function. The buffer is not filled with zeros for a version 3.*x* application.

Windows 95, OEM Service Release 2

Windows 95, OEM Service Release 2 is the most recent OEM (Original Equipment Manufacturer) release of the Windows 95 operating system. Beginning in the second half of 1996, this is the version of Windows 95 that most computer manufacturers will pre-install on new machines.

Elements of Interest to Win32 Programmers

Several elements of Windows 95, OEM Service Release 2 may be of interest to Win32 programmers:

- The FAT32 File System
- The GetDiskFreeSpaceEx Function
- The Cryptographic API
- DirectX 2
- ActiveMovie
- Internet Explorer 3.0 and the Windows Internet Extensions API
- OpenGL 1.1

The FAT32 File System

Windows 95, OEM Service Release 2 includes FAT32 File System.

FAT32 is an enhancement of the File Allocation Table (FAT) file system. FAT32 supports large drives with improved disk space efficiency.

Most Win32-based applications do not need to be revised for FAT32. The only applications that will need revising are those that use low-level disk structures, or those that use API functions that are dependent upon the on-disk FAT format of data.

The GetDiskFreeSpaceEx Function

Windows 95, OEM Service Release 2 supports the **GetDiskFreeSpaceEx** function.

Under Windows 95, including OEM Service Release 2, the existing Win32 function **GetDiskFreeSpace** caps the amount of free or total disk space reported on large drives to 2 gigabytes. Under OEM Service Release 2, in order to maintain compatibility with existing applications, even on volumes that are smaller than 2 gigabytes, **GetDiskFreeSpace** may return incorrect values.

The **GetDiskFreeSpaceEx** function obtains correct values for all volumes, including those that are larger than 2 gigabytes. New Win32-based applications should use the **GetDiskFreeSpaceEx** function. It is supported on Windows 95 OEM Service Release 2 and on Windows NT 4.0. As with all Windows 95, OEM Service Release 2 features, however, applications that call **GetDiskFreeSpaceEx** should be prepared to fall back to the **GetDiskFreeSpace** function to maintain compatibility with older Win32 implementations.

The Cryptographic API

Windows 95, OEM Service Release 2 supports the Cryptographic Application Programming Interface (CryptoAPI).

CryptoAPI is a set of functions that let applications encrypt or digitally sign data in a flexible manner, while providing protection for the user's sensitive private key data.

DirectX 2

Windows 95, OEM Service Release 2 includes DirectX 2.

DirectX 2 is a set of technologies that assist in multimedia and games programming. DirectX 2 includes DirectDraw, DirectSound, and DirectPlay.

ActiveMovie

Windows 95, OEM Service Release 2 includes ActiveMovie™.

ActiveMovie is a cross-platform digital video technology for the desktop and the Internet. ActiveMovie lets developers and creative professionals create and deliver stunning titles on multiple platforms with crisp synchronized audio, video, and special effects.

Internet Explorer 3.0 and the Windows Internet Extensions API

Windows 95, OEM Service Release 2 includes Internet Explorer 3.0, which in turn includes the Windows Internet Extensions API (WinInet).

WinInet provides a set of abstractions, layered over Windows Sockets, which support programming common operations for Internet protocols such as FTP and Gopher.

You can find documentation for WinInet in the ActiveX SDK. You can find the most current information about the ActiveX SDK at http://www.microsoft.com/intdev/.

OpenGL 1.1

Windows 95, OEM Service Release 2 includes OpenGL 1.1.

OpenGL is a three-dimensional (3-D) graphics software interface with which programmers create high-quality still and animated 3-D color images. The OpenGL overview describes the Windows NT and Windows 95 implementation of OpenGL.

Detecting Windows 95, OEM Service Release 2

Use the **GetVersionEx** function to determine that a system is running Windows 95, OEM Service Release 2 or a later release of the Windows 95 operating system.

The **GetVersionEx** function fills the members of an **OSVERSIONINFO** data structure. If the **dwPlatformId** member of that structure is VER_PLATFORM_WIN32_WINDOWS, and the low word of the **dwBuildNumber** member is greater than 1080, the system is running Windows 95,OEM Service Release 2 or a later release of Windows 95.

MS-DOS applications can check for an MS-DOS version number of 7.1 or higher to determine that a system is running Windows 95,OEM Service Release 2 or a later release of Windows 95.

Using Windows 95, OEM System Release 2 Functionality

After you have determined that a system is running Windows 95, OEM Service Release 2 or later, you can attempt to use its functionality. Some functions can be called directly. For other functions, you need to call the **LoadLibrary** or **LoadLibraryEx** function to load the appropriate DLL file, then call the **GetProcAddress** function to obtain an address for the function.

Applications that attempt to use Windows 95, OEM System Release 2 functionality should be prepared to fall back to earlier Windows 95 functionality in order to maintain compatibility with earlier versions of Windows 95.

Device I/O Control

Microsoft® Windows® 95 includes a device input and output control (IOCTL) interface that allows applications developed for the Microsoft® Win32® application programming interface (API) to communicate directly with virtual device drivers.

About Device I/O Control

Applications typically use the IOCTL interface to carry out selected Microsoft® MS-DOS® system functions, to obtain information about a device, or to carry out input and output (I/O) operations that are not available through standard Win32 functions.

This article describes the device IOCTL interface, explains how to use the interface in applications, and describes how to implement the interface in virtual devices (VxDs). For information about the device IOCTL interface for other operating systems that support the Win32 API, see the documentation included in the Microsoft Win32 Software Development Kit (SDK).

Input and Output Control in Applications

You use the device IOCTL interface in an application to carry out "low-level" operations that are not supported by the Win32 API and that require direct communication with a VxD. Windows 95 implements the interface through the **DeviceIoControl** function, which sends commands and accompanying data directly to the given VxD. To use the interface, you open the VxD by using the **CreateFile** function, send commands to the VxD by using **DeviceIoControl**, and finally close the VxD by using the **CloseHandle** function.

Opening the VxD

You can open a static or dynamically loadable VxD by specifying the module name, filename, or registry entry identifying the VxD in a call to the **CreateFile** function. If the VxD exists and it supports the device IOCTL interface, the function returns a device handle that you can use in subsequent calls to the **DeviceIoControl** function. Otherwise, the function fails and sets the last error value to ERROR_NOT_SUPPORTED or ERROR_FILE_NOT_FOUND. You can use the **GetLastError** function to retrieve the error value.

When you open a VxD, you must specify a name having the following form.

```
\\.\VxdName
```

VxDName can be the module name of the VxD, the name of the VxD file, or the name of a registry entry that specifies the filename.

CreateFile checks for a filename extension to determine whether *VxDName* specifies a file. If a filename extension (such as .VXD) is present, the function looks for the file in the standard search path. In the following example, **CreateFile** looks for the SAMPLE.VXD file in the standard search path.

```
HANDLE hDevice;

hDevice = CreateFile("\\\\.\\SAMPLE.VXD", 0, 0, NULL, 0,
    FILE_FLAG_DELETE_ON_CLOSE, NULL);
```

If *VxDName* has no filename extension, **CreateFile** checks the registry to see if the name is also a value name under the **KnownVxDs** key in **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\SessionManager**. If it is a value name, **CreateFile** uses the current value associated with the name as the full path of the VxD file. This method is useful for specifying VxDs that are not in the standard search path. In the following example, **CreateFile** searches the registry for the MYVXDPATH value.

```
hDevice = CreateFile("\\\\.\\MYVXDPATH", 0, 0, NULL, 0,
    FILE_FLAG_DELETE_ON_CLOSE, NULL);
```

If *VxDName* has no filename extension and is not in the registry, **CreateFile** assumes that the name is a VxD module name and searches the internally maintained device descriptor blocks for an already loaded VxD having the given name. In the following example, **CreateFile** opens the standard VxD named VWIN32.VXD.

```
hDevice = CreateFile("\\\\.\\VWIN32", 0, 0, NULL, 0,
    0, NULL);
```

In all cases, if **CreateFile** cannot find or load the VxD, it sets the last error value to ERROR_FILE_NOT_FOUND. If the function loads the VxD but the VxD does not support the device IOCTL interface, **CreateFile** sets the last error value to ERROR_NOT_SUPPORTED.

You can open the same VxD any number of times. **CreateFile** provides a unique handle each time you open a VxD, but it makes sure that no more than one copy of the VxD is loaded into memory. To ensure that the system removes the VxD from memory when you close the last instance of the VxD, use the FILE_FLAG_DELETE_ON_CLOSE value when opening dynamically loadable VxDs. Static VxDs cannot be removed from memory.

Although **CreateFile** has several parameters, only the *lpName* and *fdwAttrsAndFlags* parameters are useful when opening an VxD. *fdwAttrsAndFlags* can be zero, the FILE_FLAG_DELETE_ON_CLOSE value, or the FILE_FLAG_OVERLAPPED value. FILE_FLAG_OVERLAPPED is used for asynchronous

operation and is described later in this article.

Sending Commands

You use **DeviceIoControl** to send commands to a VxD. You must specify the previously opened device handle, control code, and input and output parameters for the call. The device handle identifies the VxD, and the control code specifies an action for the VxD to perform. In the following example, the DIOC_GETVERSION control code directs the given VxD to return version information.

```
HANDLE hDevice
BYTE bOutput[4];
DWORD cb;

fResult = DeviceIoControl(
    hDevice,          // device handle
    DIOC_GETVERSION,  // control code
    NULL, 0,          // input parameters
    bOutput, 4, &cb,  // output parameters
    0);
```

The input and output parameters of **DeviceIoControl** include the addresses and sizes of any buffers needed to pass data into or out of the VxD. Whether you use these parameters depends on how the VxD processes the control code. You supply an input buffer if the VxD requires that you pass it data for processing, and you supply an output buffer if the VxD returns the results of processing. In the previous example, only the output parameters are supplied. These include the address of the output buffer; the size, in bytes, of the buffer; and the address of the variable to receive the count of bytes actually copied to the buffer by the VxD.

Although the Win32 header files define a set of standard control codes, Windows 95 does not support these standard codes. Instead, the meaning and value of control codes in Windows 95 are specific to each VxD. Different VxDs may support different control codes.

Some VxDs support the DIOC_GETVERSION control code, which directs the VxD to return version information in the output buffer. Although the version information can have any format that helps the application determine the version of the VxD, keeping the information to 4 bytes or less is recommended. The VxD returns the version information only if you supply a buffer and specify a nonzero size for the buffer.

If you opened the VxD using the FILE_FLAG_OVERLAPPED value, you must also provide an **OVERLAPPED** structure when calling **DeviceIoControl**. This structure contains information that the VxD uses to process the control code asynchronously.

Closing a VxD

When you have finished using a VxD, you can close the associated device handle by using the **CloseHandle** function, or you can let the operating system close the handle when the application terminates. The following example closes a VxD.

```
CloseHandle(hDevice);
```

Closing a VxD does not necessarily remove the VxD from memory. If you open a dynamically loadable VxD using the FILE_FLAG_DELETE_ON_CLOSE value, **CloseHandle** also removes the VxD if no other valid handles are present in the system. The system maintains a reference count for dynamically loadable VxDs, incrementing the count each time the VxD is opened and decrementing when the VxD is closed. **CloseHandle** checks this count and removes the VxD from memory when the count reaches zero. The system does not keep a reference count for static VxDs; it does not remove these VxDs when their corresponding handles are closed.

In rare cases, you may need to use the **DeleteFile** function to remove a dynamically loadable VxD from memory. For example, you use **DeleteFile** if another application has loaded the VxD and you just want to unload it. You also use **DeleteFile** if you have successfully loaded a VxD by using **CreateFile**, but the VxD does not support the device IOCTL interface. In such cases, **CreateFile** loads the VxD but provides no handle to close and remove the VxD. The following example removes the VxD named SAMPLE from memory.

```
DeleteFile("\\\\.\\SAMPLE");
```

In this example, SAMPLE is the module name of the VxD. (Do not specify the filename.) Be aware that the module name of a VxD is not necessarily the same as the VxD's filename without a filename extension. In general, avoid using **DeleteFile** to remove a VxD from memory.

Asynchronous Operations and VxDs

You can direct a VxD to process a control code asynchronously. In an asynchronous operation, the **DeviceIoControl** function returns immediately, regardless of whether the VxD has finished processing the control code. Asynchronous operation allows an application to continue while the VxD processes the control code in the background. You request an asynchronous operation by specifying the address of an **OVERLAPPED** structure in the **DeviceIoControl** function. The **hDevice** member of **OVERLAPPED** specifies the handle of an event that the system sets to the signaled state when the VxD has completed the operation.

Asynchronous (overlapped) operations are useful for lengthy operations, such as formatting a disk. To perform an asynchronous operation, you must specify the FILE_FLAG_OVERLAPPED value when calling **CreateFile** to obtain a device handle. When calling **DeviceIoControl**, you must specify the address of an **OVERLAPPED** structure in the *lpOverlapped* parameter and the handle of a manual reset event in the **hEvent** member of the structure. The system ignores all other members.

If **DeviceIoControl** completes the operation before returning, it returns TRUE; otherwise, it returns FALSE. When the operation is finished, the system signals the manual reset event. You should call **GetOverlappedResult** when the thread that called **DeviceIoControl** needs to wait (that is, stop executing) until the operation has finished.

Using VWIN32 to Carry Out MS-DOS Functions

Windows 95 provides a VxD named VWIN32.VXD that supports a set of control codes that Win32-based applications can use to carry out selected MS-DOS system functions. These system-defined control codes consist of the following values.

Control code (value)	Meaning
VWIN32_DIOC_DOS_DRIVEINFO (6)	Performs Interrupt 21h Function 730X commands. This value is supported in Windows 95 OEM Service Release 2 and later.
VWIN32_DIOC_DOS_INT13 (4)	Performs Interrupt 13h commands
VWIN32_DIOC_DOS_INT25 (2)	Performs the Absolute Disk Read command (Interrupt 25h)
VWIN32_DIOC_DOS_INT26 (3)	Performs the Absolute Disk Write command (Interrupt 25h)
VWIN32_DIOC_DOS_IOCTL (1)	Performs the specified MS-DOS device I/O control function (Interrupt 21h Function 4400h through 4411h)

When an application calls the **DeviceIoControl** function with the *dwIoControlCode* parameter set to one of the predefined control codes, the *lpvInBuffer* and *lpvOutBuffer* parameters must specify the addresses of **DIOC_REGISTERS** structures. The **DIOC_REGISTERS** structure specified by *lpvInBuffer* contains a set of register values that specify a command for the VxD to execute and any data that the VxD needs to execute the command. After completing the command, the VxD fills the **DIOC_REGISTERS** structure specified by *lpvOutBuffer* with the register values that resulted from executing the command. The meaning of the register values depends on the specified command. Many of the MS-DOS and BIOS functions require segment:offset register pairs for pointers. However, because 32-bit code does not have segments, the **DIOC_REGISTERS** structure does not contain members for segment registers. Place the full pointer that would go into a real-mode segment:offset register pair into the structure member that corresponds to the offset of the register pair. For example, use the **reg_EDI** member for the pointer that would go into the ES:DI registers. When an application uses the **DeviceIoControl** function to send commands to a VxD other than VWIN32.VXD, the meaning of the function's parameters are defined by the VxD. The system does not validate the parameters. The system VxD, VWIN32.VXD, supports the IOCTL functions originally provided by MS-DOS Interrupt 21h. The following example shows how to call Get Media ID (Interrupt 21h Function 440Dh Minor Code 66h) from a Win32-based application.

```
#define VWIN32_DIOC_DOS_IOCTL 1

typedef struct _DIOC_REGISTERS {
    DWORD reg_EBX;
    DWORD reg_EDX;
    DWORD reg_ECX;
    DWORD reg_EAX;
    DWORD reg_EDI;
    DWORD reg_ESI;
    DWORD reg_Flags;
} DIOC_REGISTERS, *PDIOC_REGISTERS;

// Important: All MS_DOS data structures must be packed on a
// one-byte boundary.
```

```
#pragma pack(1)
typedef struct _MID {
    WORD  midInfoLevel;
    DWORD midSerialNum;
    BYTE  midVolLabel[11];
    BYTE  midFileSysType[8];
} MID, *PMID;
#pragma pack()

HANDLE hDevice;
DIOC_REGISTERS reg;
MID mid;
BOOL fResult;
DWORD cb;
int nDrive = 3;  // Drive C:

hDevice = CreateFile("\\\\.\\vwin32",
    0, 0, NULL, 0, FILE_FLAG_DELETE_ON_CLOSE, NULL);

reg.reg_EAX = 0x440D;       // IOCTL for block devices
reg.reg_EBX = nDrive;       // zero-based drive identifier
reg.reg_ECX = 0x4866;       // Get Media ID command
reg.reg_EDX = (DWORD) &mid; // receives media identifier information
reg.reg_Flags = 0x0001;     // assume error (carry flag is set)

fResult = DeviceIoControl(hDevice,
    VWIN32_DIOC_DOS_IOCTL,
    &reg, sizeof(reg),
    &reg, sizeof(reg),
    &cb, 0);

if (!fResult || (reg.reg_Flags & 0x0001))
    ; // error if carry flag is set

CloseHandle(hDevice);
```

Supporting Input-Output Control in VxDs

A VxD can support the device IOCTL interface by processing the W32_DEVICEIOCONTROL message in the VxD's control procedure. You can take advantage of the device IOCTL interface by providing a VxD that performs privileged (ring 0) operations for your application.

Loading and Opening the VxD

When your application calls **CreateFile**, the system sends the W32_DEVICEIOCONTROL message to the control procedure of the specified VxD to determine if it supports the device IOCTL interface. The ESI register contains the address of a **DIOCParams** structure whose **dwIoControlCode** member specifies the DIOC_GETVERSION control code. A VxD that supports the device IOCTL interface must respond to the DIOC_GETVERSION control code by clearing the EAX register. (A VxD can also return version information in the buffer pointed to by the **lpvOutBuffer** member so long as the **cbOutBuffer** member is nonzero. If **cbOutBuffer** is zero, the calling application is not interested in version information, but the VxD must still return zero to indicate success.) If the VxD does not support the device IOCTL interface, it must place a nonzero value in EAX.

For dynamically loadable VxDs, the system sends a SYS_DYNAMIC_INIT control message to the VxD the first time that it is opened. If the VxD returns success, the system sends the W32_DEVICEIOCONTROL message with the DIOC_OPEN (identical in value to the DIOC_GETVERSION control code) control code. The VxD must return zero to inform the calling application that it supports W32_DEVICEIOCONTROL. The system sets the reference count for the VxD to 1. On every subsequent call to **CreateFile** for the VxD, the VxD receives the W32_DEVICEIOCONTROL message with the DIOC_OPEN control code and the reference count for the VxD is incremented.

The VxD receives a W32_DEVICEIOCONTROL message with the DIOC_CLOSEHANDLE control code if the application closes the device handle by calling the **CloseHandle** function, or if the operating system closes the handle when the application terminates (or **DeleteFile** if the VxD was not opened with the FILE_FLAG_DELETE_ON_CLOSE value). The VxD can use this notification to perform cleanup operations and release structures associated with the application. The reference count for the VxD is decremented before the message is sent. If the reference count is decremented to zero, the VxD receives the SYS_DYNAMIC_EXIT message and is subsequently unloaded.

Processing Control Codes

When an application calls **DeviceIoControl**, the system calls the control procedure of the VxD identified by the given device handle. The EAX register contains the W32_DEVICEIOCONTROL message, and the ESI register contains the address of a **DIOCParams** structure. The structure contains all of the parameters that the application specified in the **DeviceIoControl** function as well as additional information. The VxD should examine the **dwIoControlCode** member of the **DIOCParams** structure to determine the action to perform. The **lpvInBuffer** member contains supporting data that the VxD needs to complete the action. After processing the control code, the VxD should copy any information that it needs to return to the application to the buffer specified by the **lpvOutBuffer** member.

If the VxD successfully processes the control code, it should clear the EAX register before returning. Otherwise, the VxD should set EAX to a nonzero value.

Checking for Asynchronous Operations

A VxD can determine whether an application has requested an asynchronous operation by checking the **lpoOverlapped** member of the **DIOCParams** structure. If **lpoOverlapped** specifies the address of an **OVERLAPPED** structure, the application has requested an asynchronous operation. In that case, the VxD should return - 1 in the EAX register and then process the specified control code.

The operating system does not take any steps to make the application's memory available to the VxD at all times (in all contexts) for asynchronous operations. When implementing asynchronous operations, the VxD must use the appropriate virtual machine manager (VMM) services, such as **LinPageLock** with the PAGEMAPGLOBAL value, to make the application's memory pages available across contexts, including access to the **OVERLAPPED** structure.

When the VxD finishes processing the control code, it must notify the application by calling the **VWIN32_DIOCCompletionRoutine** service provided by VWIN32.VXD. The EBX register must contain the value of the **Internal** member of the **OVERLAPPED** structure. This member is reserved for operating system use only. Also, if the VxD copies any data to the buffer specified by the **lpvOutBuffer** member of the **DIOCParams** structure, the VxD must specify the count of bytes copied to the buffer in the **InternalHigh** member of **OVERLAPPED**. The remaining members of **OVERLAPPED**, **Offset** and **OffsetHigh**, can be used for developer-defined data.

Essentially, the **VWIN32_DIOCCompletionRoutine** service helps signal the event identified by the **hEvent** member of the **OVERLAPPED** structure. The application monitors the event to determine when the asynchronous operation is completed.

Device I/O Control Reference

The following structures, system messages, and service are associated with the device IOCTL interface.

Device I/O Control Structures

These structures are used to carry out device IOCTL functions:

DIOC_REGISTERS
DIOCParams

Device I/O Control System Messages

The following system message is used to implement the device IOCTL interface of a VxD:

W32_DEVICEIOCONTROL

Device I/O Services

The following service is used with device IOCTL:

VWIN32_DIOCCompletionRoutine

Exclusive Volume Locking

This article describes exclusive volume locking and provides guidelines for applications that carry out direct access on volumes while running with Microsoft® Windows® 95. This article also describes the input and output control (IOCTL) functions that applications need for managing exclusive volume locking.

About Exclusive Volume Locking

Disk utilities and other applications that directly modify file system structures, such as directory entries, must request exclusive volume locking (that is, exclusive use of the volume) before making modifications to the structures. Exclusive use prevents applications from inadvertently changing the file system while a disk utility is trying modify it and ensures that the information on a volume represents the current state of the volume.

Direct Access

Applications typically use direct access to make changes to the directory entries, paths, and allocation chains in the file allocation table (FAT) of a given volume. The applications access this information by using the Interrupt 13h functions, Absolute Disk Read and Write (Interrupts 25h and 26h), or the Interrupt 21h read, write, and format track IOCTL functions. Some applications may also access the volume using the disk controller input and output (I/O) ports.

As a multitasking operating system, Windows 95 permits any number of applications to access a volume at a given time. Applications that change the file system structures of a volume without regard to other applications risk corruption of that information and subsequent data loss. To prevent data loss, the system manages all requests for direct access. With the exception of floppy disk drives, Windows 95 does not permit direct write operations, unless the volume has been locked by the application. The system returns the ERROR_WRITE_PROTECT error value to functions or interrupts that attempt direct write operations when the volume has not been locked. An application can read directly from a volume, but the system may satisfy the request by reading from internal caches rather than from the medium itself. An application cannot access disk controller I/O ports; the system traps all access.

An application can override the default behavior of the system by requesting exclusive use of the volume using the lock and unlock volume IOCTL functions. An application that has exclusive use of a volume can read and write directly to the medium, and because the system flushes internal caches to the medium, the information there reflects the actual state of the volume. Locking a volume (including the floppy disk drives) ensures consistency of data, because other processes cannot update information about the volume while it is locked.

Exclusive Use Lock

To request exclusive use of a volume, an application use either Lock Logical Volume (Interrupt 21h Function 440Dh Minor Code 4Ah) or Lock Physical Volume (Interrupt 21h Function 440Dh Minor Code 4Bh). Before issuing an Interrupt 13h function, an application must acquire a physical volume lock. An application that only modifies logical volumes should acquire a logical volume lock. The calling interface for both locks is the same except for how the volume to lock is specified; the physical lock requires an Interrupt 13h device unit number, but the logical lock requires a logical drive number. When an application obtains a lock on a physical drive, the system acquires a logical volume lock for each logical volume on the physical drive. Obtaining a lock on a logical volume that is the parent (or host) drive also locks the child drives. For example, a compressed volume is locked when its host drive is locked.

The application that owns the lock can carry out direct disk write operations. Only one application at a time can lock the volume. If an application already owns the volume lock, the system fails subsequent calls to lock the volume.

The volume-locking functions allow the lock owner to control the kind of access that other processes have to the volume. There are three categories of access: *read operations*, which include opening a file as well as reading from a file; *write operations*, which include deleting and renaming a file as well as writing to a file; and *new file mappings*.

When an application calls either of the lock volume functions, it specifies a *lock level* and, depending on the level, passes in additional information known as *permissions*. The lock level and permissions specify what kind of operations the system allows processes other than the lock owner to do on the volume while it is locked.

When another process attempts to access the volume, the file system will *fail* the operation and return the ERROR_WRITE_PROTECT error value or it will block the operation, depending on the lock level and permissions. The system queues a blocked operation and puts the process requesting the operation to sleep until the lock is released and the operation can be performed.

When an application has completed its work, it must unlock the volume. Depending on whether it acquired a logical or physical lock, the application unlocks the volume by calling either Unlock Logical Volume (Interrupt 21h Function 440Dh Minor Code 6Ah) or Unlock Physical Volume (Interrupt 21h Function 440Dh Minor Code 6Bh). Unlocking the volume lets the system perform any blocked operations (in the order that they occurred) and resume normal activity. If an application exits without releasing the lock, the system automatically releases it.

If an application has a lock and the user attempts to close the virtual machine in which the application is running, the system displays a message warning the user that closing the virtual machine could result in damage to the volume. If the user confirms the closing, the system releases the lock and closes the virtual machine.

An application can lock volumes on local drives, but not on network drives or on drives that are not managed by the I/O supervisor, a virtual device driver. Also, the lock and unlock volume IOCTL functions are not available in previous versions of Microsoft® MS-DOS®. If the functions are used with a previous version, they return an error value.

Windows 95 provides 4 levels of exclusive volume locks. The level 0 lock is used alone, and lock levels 1, 2, and 3 form a hierarchy that increasingly restricts access to the file system based on the permissions set when the application obtains the level 1 lock. Although the level 0 lock is not part of the locking hierarchy formed by lock levels 1, 2, and 3, it has a more restrictive sublevel for applications that format volumes. Applications should perform direct disk write operations *only* in either a level 0 or 3 lock.

Level 0 Lock

The level 0 lock cannot be obtained on volumes that have any open files or handles. This restriction includes handles returned by the Windows **FindFirstFile** and **FindFirstChangeNotification** functions. Because the system always has open files, an application cannot take a level 0 lock on the drive that contains the Windows 95 system files. The locking hierarchy can be used on volumes with open files and handles. If the system fails a level 0 lock request because of open files, an application can obtain a list of all the open files on the volume by calling Enumerate Open Files (Interrupt 21h Function 440Dh Minor Code 6Dh).

Before returning from the call to lock the volume, Windows 95 flushes to disk all data from the file system cache. To ensure that the disk and file system remain in a consistent state, the system puts the cache into write-through mode so that it can immediately commit to disk all data from file write operations. If an application has obtained a lock on a child volume, the system does not automatically flush the file system cache after a write operation to the parent volume. To ensure that the cache is flushed when opening files with Open or Create File (Interrupt 21h Function 716Ch), an application should specify the OPEN_FLAGS_NO_BUFFERING (0100h) value. If a file has not been opened with OPEN_FLAGS_NO_BUFFERING or Absolute Disk Write (Interrupt 26h) has been used to write to the parent volume, an application should call Reset Drive (Interrupt 21h Function 710Dh) to flush the cache.

After a process obtains a level 0 lock, the system only allows the lock owner to have access to the volume. The system fails all read operations, write operations, and new file mappings by other processes until the lock owner releases the lock. An application should use a level 0 lock whenever possible, because it guarantees that no other process can access the locked volume.

An application that formats volumes must obtain a more restrictive mode of the level 0 lock. To format a volume, the application should follow these steps:

1. Call the lock volume function to obtain a level 0 lock.
2. Do any needed file system I/O that uses the Interrupt 21h file handle I/O functions while in the level 0 lock.
3. Call the lock volume function a second time to obtain the more restrictive level 0 lock for formatting. When calling the lock volume function the second time, an application should specify 0 as the lock level and specify 4 in the permissions. The application, however, must already own the level 0 lock to obtain this lock.
4. Format the disk using the Interrupt 21h IOCTL functions (such as Read/Write/Format/Verify Track on Logical Drive or Get/Set Device Parameters). Interrupt 13h will work too, but it is not the preferred method for performance reasons. The system allows direct disk I/O, but it fails all other file system I/O that uses the Interrupt 21h file handle I/O functions while the restrictive level 0 lock is in effect.
5. Release the level 0 lock for formatting. An application is still in a level 0 lock at this point. Before the application releases this lock, the disk must be in a state that would be recognizable as normal FAT media. Otherwise, the functions mentioned in step 6 will not work.
6. Resume normal file system I/O using the Interrupt 21h file handle I/O functions.
7. Release the level 0 lock.

Locking Hierarchy

The locking hierarchy allows an application to obtain a lock in preparation for modifying the file system and yet allows other processes access to the drive. In this way, other processes are only denied access to the volume when absolutely necessary.

An application should perform direct disk write operations only within a level 3 lock. To obtain a level 3 lock, an application must make three calls to the lock volume function – first to obtain a level 1 lock, then a level 2 lock, and finally a level 3 lock. After obtaining the level 3 lock, the application can safely access the volume directly.

To release the lock on a volume, an application must call the appropriate unlock volume function the same number of times that the corresponding lock volume function was called. Each call to the unlock volume function decrements the lock level. For example, a level 3 lock returns to a level 2 lock, and the system processes any blocked read operations. A level 2 lock returns to a level 1 lock, and the system processes any blocked write operations or new file mappings. A call to the unlock volume function on a level 1 lock, however, releases the lock on the volume and allows other processes to obtain the lock.

Exclusive volume locks are owned by a process, not a thread. If necessary, a multithreaded application can obtain a level 1 lock in one thread, a level 2 lock in another, and a level 3 lock in yet another.

Even though the system may block or fail new file mappings, other processes are allowed to write to files through existing file mappings, because file mappings cannot be resized. Writing to an existing file mapping only changes the contents of the memory-mapped file, it does not cause changes to the file system.

Level 1

The level 1 lock acts as a sentinel to guarantee that only one application may obtain the level 2 and 3 locks. An application specifies permissions only when requesting the level 1 lock. The lock level and permissions determine the kind of access that processes other than the lock owner have to the volume while it is locked. Bit 0 of the *Permissions* parameter for the lock volume function determines if the system allows or fails write operations by other processes. Bit 1 of *Permissions* determines if the system allows or fails new file mappings by other processes. Bit 2 of *Permissions* is only used when obtaining the restrictive level 0 lock for formatting and is ignored for a level 1 lock. The following table shows which operations are allowed at each lock level based on the permissions set on the level 1 lock.

Permissions	**Level 1**	**Level 2**	**Level 3**
Bit 0 = 0 Bit 1 = 0	Write operations are failed. New file mappings are allowed. Read operations are allowed.	Write operations are failed. New file mappings are allowed. Read operations are allowed.	Write operations are failed. New file mappings are blocked. Read operations are blocked.
Bit 0 = 0 Bit 1 = 1	Write operations are failed. New file mappings are failed. Read operations are allowed.	Write operations are failed. New file mappings are failed. Read operations are allowed.	Write operations are failed. New file mappings are failed. Read operations are blocked.
Bit 0 = 1 Bit 1 = 0	Write operations are allowed. New file mappings are allowed. Read operations are allowed.	Write operations are blocked. New file mappings are allowed. Read operations are allowed.	Write operations are blocked. New file mappings are blocked. Read operations are blocked.
Bit 0 = 1 Bit 1 = 1	Write operations are allowed. New file mappings are failed. Read operations are allowed.	Write operations are blocked. New file mappings are failed. Read operations are allowed.	Write operations are blocked. New file mappings are failed. Read operations are blocked.

Calling an unlock physical or logical volume function on a level 1 lock completely releases the lock on the volume.

Level 2

The level 2 lock prevents all processes except the lock owner from writing to the disk, but the lock allows any application to read from it. Depending on the permissions set when the application obtained the level 1 lock, the system will either block or fail write operations and either allow or fail new file mappings. Before obtaining a level 3 lock, an application should call Get Lock Flag State (Interrupt 21h Function 440Dh Minor Code 6Ch) to determine if anything on the disk has changed, such as the swap file growing or shrinking. Calling Get Lock Flag State at this point is an optimization that is done to avoid obtaining the level 3 lock unnecessarily.

Calling an unlock physical or logical volume function on a level 2 lock decrements the lock level to 1 and causes the system to perform previously blocked operations that are allowed at the lower lock level.

Level 3

The level 3 lock prevents all processes except the lock owner from reading or writing to the disk. Read operations are blocked, and write operations and new file mappings are either blocked or failed depending on the permissions set in the level 1 lock. This is the most restrictive lock, not only because it prevents all other processes from accessing the disk but also because the lock owner is limited in what it can do.

Because read operations are blocked in the level 3 lock, an application must not execute any user interface or screen update functions, spawn applications, load dynamic-link libraries (DLLs), or yield to avoid deadlock. For example, if an application yields to a process with a discardable segment that the system has discarded, the system cannot reload the discarded segment because the process cannot read from the disk. This situation results in deadlock. Whenever a process obtains a level 3 lock, it should keep the lock for as short a period as possible to avoid severely degrading system performance. The purpose of the level 3 lock is to write changes to disk, so processes should only call disk I/O functions inside the lock.

After a level 3 lock is obtained, the file system takes several steps to allow a process to write directly to the disk. First, the system flushes all file system buffers and caches. Next, it puts the cache into write-through mode so that changes will be written to disk immediately. Finally, all open files are closed at the file system driver (FSD) level, which is invisible to all processes. While an application is in a level 3 lock, the system does not permit the swap file to grow or shrink, but it can still be read from or written to.

If an application has obtained a lock on a child volume, the system does not automatically flush the file system cache after a write operation to the parent volume. To ensure that the cache is flushed when Open or Create File (Interrupt 21h Function 716Ch) is used, the application should specify the OPEN_FLAGS_NO_BUFFERING (0100h) value. If the file has not been opened with OPEN_FLAGS_NO_BUFFERING or Absolute Disk Write (Interrupt 26h) has been used to write to the parent volume, the application should call Reset Drive (Interrupt 21h Function 710Dh) to flush the cache.

Before releasing the level 3 lock, the process is responsible for putting the file system into a state consistent with what existed before the lock was obtained. The process must be careful to correctly update all file system data, such as the FAT or directory entries. If a file was opened, it must not be deleted, renamed, or moved to a different volume. Otherwise, the system can become very unstable. An application should use Enumerate Open Files (Interrupt 21h Function 440Dh Minor Code 6Dh) to obtain a list of all open files on the volume.

When the level 3 lock is released, the system unblocks all pending read operations, reopens closed files on demand, and puts the cache back into write-behind mode. The lock owner returns to a level 2 lock.

Using the Locking Hierarchy

An application should obtain a level 1 lock before beginning an operation, such as a complete defragmentation or compression. The application should release the level 1 lock only after the entire operation is finished. This approach prevents other processes from obtaining a lock on the same disk, which would keep the lock owner from finishing its work.

To minimize the time spent in the level 3 lock, a process should remain in the level 2 lock to perform certain tasks, such as computing disk statistics and preparing data packets to be written before actually writing them in the level 3 lock. As soon as a process enters the level 3 lock, the application must call Get Lock Flag State (Interrupt 21h Function 440Dh Minor Code 6Ch) to determine if anything on the disk has changed, such as the swap file growing or shrinking. If a change has occurred, the process should release the level 3 lock, return to the level 2 lock to recompute any needed information, and then obtain the level 3 lock again. If Get Lock Flag State shows that the disk has not changed, the process should do its writing and then release the level 3 lock.

Swap File

The system pager requires access to the swap file at all times, even when an application has locked the volume containing the file. To ensure access, the system always gives the system pager the opportunity to accept or reject a lock request.

If the system grants a lock to a process, the requesting process must ensure that data for the swap file remains unchanged and that the pager can safely read from or write to the swap file at any time. In particular, the process must *not* change the allocation chain of the swap file, and if it moves the swap file's directory entry, it must ensure that the path to it is always consistent. Failure to observe these guidelines can result in a system crash.

An application can determine the directory entry and allocation chain of the swap file by retrieving the path of the file using Find Swap File (Interrupt 21h Function 440Dh Minor Code 6Eh).

The system permits the swap file to grow or shrink in the level 1 and 2 locks. Because of this, an application should call Get Lock Flag State (Interrupt 21h Function 440Dh Minor Code 6Ch) after obtaining a level 3 lock to determine if anything on the disk has changed as the result of the swap file growing or shrinking.

Lock Requests and Virtual Devices

The system broadcasts a message to all virtual devices (VxDs) when an application makes a request for a lock. Any VxD that uses virtual block device and I/O supervisor services to carry out direct I/O on the given volume must check for this message and either accept or reject the request when the message is received.

If the VxD rejects the request, it must return the appropriate error value. In such cases, the system does not grant the lock to the requesting application. If a VxD accepts the request, it must avoid all operations that may affect the consistency of the volume for the duration of the lock.

When the application releases the lock, the system again issues a broadcast message. The VxD can resume normal operation at this point.

Volume-Locking Guidelines

Applications that lock and modify volumes should follow these guidelines to avoid degrading system performance and to prevent data loss:

- If there are no open files on the volume, applications should perform direct disk writes in a level 0 lock. Otherwise, they should use the locking hierarchy and perform disk write operations in a level 3 lock.
- Applications should utilize the locking hierarchy to minimize the time spent in a level 3 lock. They should only call disk I/O functions inside a level 3 lock and drop down to a level 1 or 2 lock whenever possible.
- Applications should neither terminate or relinquish control nor leave a level 0 or 3 lock if the volume information is incomplete or invalid. When applications leave one of these locks, the file system *must* be consistent with what it was when they entered the lock because other applications will regain access to the drive.
- Because the Interrupt 21h file handle I/O functions rely on accurate information about the volume, applications should not use these functions when the volume information is incomplete or invalid.
- Applications should not move the swap file.
- Applications should not move memory-mapped files opened for write access. Read-only memory-mapped files may be moved cluster by cluster.
- Applications may only move 32-bit Windows-based DLLs and executables cluster by cluster.
- Applications may move directory entries for the swap file and open memory-mapped files, but the path to them must always be consistent, even in a level 3 lock.

Because read operations are blocked in the level 3 lock, all applications written for 16-bit Windows, 32-bit Windows, or MS-DOS should follow these guidelines to avoid deadlock while in a level 3 lock:

- Applications should only access the disk by using the low-level disk functions (Interrupt 13h, Interrupt 25h, and Interrupt 26h) or the Interrupt 21h file handle read, write, seek, and IOCTL functions. Other MS-DOS functions are not guaranteed to work. Windows or C run-time library file I/O functions should not be used, because these functions may contain code or call code that is not safe to execute inside the level 3 lock.
- Applications should not yield control, update the screen, execute any user-interface code, or do anything else that could cause Windows 95 to load a new or previously discarded segment, such as by spawning an application or loading a DLL.
- Windows-based applications must have all the code for a level 3 lock contained within the processing for a single message. The application should not process other messages or call any Windows functions.

Special Considerations for 32-bit Windows-Based Applications

32-bit Windows-based applications must call the exclusive volume-locking IOCTL functions indirectly by opening VWIN32.VXD and using its **DeviceIoControl** interface.

In response, VWIN32.VXD issues the low-level disk I/O functions (Interrupt 13h, Interrupt 25h, and Interrupt 26h) as well as the MS-DOS Interrupt 21h file handle read, write, seek, and IOCTL functions in the context of the calling process.

32-bit Windows-based applications may safely call the Windows **ReadFile**, **WriteFile**, **SetFilePointer**, and **DeviceIoControl** functions within a level 3 lock. Other Windows or C run-time library functions should not be used while in the level 3 lock, because these functions may call other functions that are not safe inside a level 3 lock.

Special Considerations for 16-bit Windows-Based Applications

16-bit Windows-based applications may call the exclusive volume-locking IOCTL functions directly using the Interrupt 21h interface, which is described in the reference material. These applications must mark all of the code, data, and resource segments that will be accessed while inside the level 2 and 3 locks as PRELOAD NONDISCARDABLE in the module-definition (.DEF) file. This marking prevents deadlock in case the system needs to load one of the application's segments from the executable file inside a level 3 lock.

Special Considerations for MS- DOS - Based Applications

MS-DOS - based applications may directly call the volume-locking IOCTL functions by using Interrupt 21h, as described in the reference material.

When a windowed MS-DOS - based application obtains a level 3 lock, the system forces it to full screen mode to avoid deadlock with the display driver. When the application releases the level 3 lock, it remains in full screen mode.

Applications must not call the Advanced SCSI Programming Interface (ASPI) functions inside a level 3 lock. These functions bypass the file system, leaving it in an inconsistent state.

When MS-DOS - based applications run in single MS-DOS application mode (real mode), they may issue the volume-locking IOCTL functions and the functions will succeed. However, because there is no multitasking, there is only one lock rather than a hierarchy as when Windows 95 is running. The volume-locking IOCTL functions will fail on down-level versions of MS-DOS.

Exclusive Volume Locking Reference

The following functions can be used to manage exclusive volume locking.

Lock Logical Volume	Interrupt 21h Function 440Dh Minor Code 4Ah
Lock Physical Volume	Interrupt 21h Function 440Dh Minor Code 4Bh
Unlock Logical Volume	Interrupt 21h Function 440Dh Minor Code 6Ah
Unlock Physical Volume	Interrupt 21h Function 440Dh Minor Code 6Bh
Get Lock Flag State	Interrupt 21h Function 440Dh Minor Code 6Ch
Enumerate Open Files	Interrupt 21h Function 440Dh Minor Code 6Dh
Find Swap File	Interrupt 21h Function 440Dh Minor Code 6Eh
Get Current Lock State	Interrupt 21h Function 440Dh Minor Code 70h
Reset Drive	Interrupt 21h Function 710Dh

The exclusive-volume locking IOCTL functions are similar to other MS-DOS functions. An application must copy function parameters to registers and issue an Interrupt 21h instruction to carry out the call.

FAT32 File System

FAT32 is a derivative of the File Allocation Table (FAT) file system that supports drives with over 2GB of storage. Because FAT32 drives can contain more than 65,526 clusters, smaller clusters are used than on large FAT16 drives. This method results in more efficient space allocation on the FAT32 drive.

File System Specifications

The largest possible file for a FAT32 drive is 4GB minus 2 bytes. Win32-based applications can open files this large without special handling. However, non-Win32 applications must use **Int 21h Function 6Ch (FAT32)** with the EXTENDED_SIZE flag.

The FAT32 file system includes four bytes per cluster within the file allocation table. This differs from the FAT16 file system, which contains 2 bytes per cluster, and the FAT12 file system, which contains 1.5 bytes per cluster within the file allocation table.

Note that the high 4 bits of the 32-bit values in the FAT32 file allocation table are reserved and are not part of the cluster number. Applications that directly read a FAT32 file allocation table must mask off these bits and preserve them when writing new values.

File System Cluster Limits:

System	Cluster Limit
FAT12	The count of data clusters is less than 4087 clusters.
FAT16	The count of data clusters is between 4087 and 65526 clusters, inclusive.
FAT32	The count of data clusters is greater than 65526 clusters.

Boot Sector and Bootstrap Modifications

Reserved Sectors

FAT32 drives contain more reserved sectors than FAT16 or FAT12 drives. The number of reserved sectors is usually 32, but can vary.

Boot Sector Modifications

Because a FAT32 BIOS Parameter Block (BPB), represented by the **BPB (FAT32)** structure, is larger than a standard BPB, the boot record on FAT32 drives is greater than 1 sector. In addition, there is a sector in the reserved area on FAT32 drives that contains values for the count of free clusters and the cluster number of the most recently allocated cluster. These values are members of the **BIGFATBOOTFSINFO (FAT32)** structure which is contained within this sector. These additional fields allow the system to initialize the values without having to read the entire file allocation table.

Root Directory

The root directory on a FAT32 drive is not stored in a fixed location as it is on FAT16 and FAT12 drives. On FAT32 drives, the root directory is an ordinary cluster chain. The **A_BF_BPB_RootDirStrtClus** member in the **BPB (FAT32)** structure contains the number of the first cluster in the root directory. This allows the root directory to grow as needed. In addition, the **BPB_RootEntries** member of **BPB (FAT32)** is ignored on a FAT32 drive.

Sectors Per FAT

The **A_BF_BPB_SectorsPerFAT** member of **BPB (FAT32)** is *always* zero on a FAT32 drive. Additionally, the **A_BF_BPB_BigSectorsPerFat** and **A_BF_BPB_BigSectorsPerFatHi** members of the updated **BPB (FAT32)** provide equivalent information for FAT32 media.

FAT Mirroring

On all FAT drives, there may be multiple copies of the FAT. If an error occurs reading the primary copy, the file system will attempt to read from the backup copies. On FAT16 and FAT12 drives, the first FAT is always the primary copy and any modifications will automatically be written to all copies. However, on FAT32 drives, FAT mirroring can be disabled and a FAT other than the first one can be the primary (or "active") copy of the FAT.

Mirroring is enabled by clearing bit 0x0080 in the **extdpb_flags** member of a FAT32 Drive Parameter Block (DPB) structure, **DPB (FAT32)**.

When Enabled (bit 0x0080 clear)

With mirroring enabled, whenever a FAT sector is written, it will also be written to every other FAT. Also, a mirrored FAT sector can be read from any FAT.

A FAT32 drive with multiple FATs will behave the same as FAT16 and FAT12 drives with multiple FATs. That is, the multiple FATs are backups of each other.

When Disabled (bit 0x0080 set)

With mirroring disabled, only one of the FATs is active. The active FAT is the one specified by bits 0 through 3 of the **extdpb_flags** member of **DPB (FAT32)**. The other FATs are ignored.

Disabling mirroring allows better handling of a drive with a bad sector in one of the FATs. If a bad sector exists, access to the damaged FAT can be completely disabled. Then, a new FAT can be built in one of the inactive FATs and then made accessible by changing the active FAT value in **extdpb_flags**.

Partition Types

The following are all the valid partition types and their corresponding values for use in the **Part_FileSystem** member of the **s_partition (FAT32)** structure.

Partition Types

Value	Description
PART_UNKNOWN(00h)	Unknown
PART_DOS2_FAT(01h)	12-bit FAT
PART_DOS3_FAT(04h)	16-bit FAT. Partitions smaller than 32MB.
PART_EXTENDED(05h)	Extended MS-DOS Partition
PART_DOS4_FAT(06h)	16-bit FAT. Partitions larger than or equal to 32MB.
PART_DOS32(0Bh)	32-bit FAT. Partitions up to 2047GB.
PART_DOS32X(0Ch)	Same as PART_DOS32(0Bh), but uses Logical Block Address Int 13h extensions.
PART_DOSX13(0Eh)	Same as PART_DOS4_FAT(06h), but uses Logical Block Address Int 13h extensions.
PART_DOSX13X(0Fh)	Same as PART_EXTENDED(05h), but uses Logical Block Address Int 13h extensions.

Limitations of Some Existing Functions

The following functions have limitations on FAT32 media:

File Control Block Functions

The **open** and **create** functions will only work for creating a volume label on a FAT32 drive. However, FCB Find_First/Next, Delete, and Rename will work and will return the entire 32-byte directory entry on FAT32 media.

Handle-Based File Write Functions

Win32-based applications can extend a file to a size greater than 2GB minus 1byte without special handling. However, non-Win32 applications must open such files with **Int 21h Function 6Ch (FAT32)** specifying the EXTENDED_SIZE flag. If not, write functions will fail and ERROR_ACCESS_DENIED (0005h) will be returned.

Handle-Based File Open and Create Functions

Win32-based applications can open files larger than 2GB minus 1byte in size without special handling. Non-Win32-based applications must open such files with **Int 21h Function 6Ch (FAT32)** specifying the EXTENDED_SIZE flag. However, using an old style create to truncate a file will not fail.

FAT32 File System Reference

The following functions and structures are supported on the FAT32 file system.

FAT32 File System Functions

The FAT32 file system supports the following new Int 21h functions and updated IOCTL functions.

New Int 21h Functions

Several new interrupt 21h functions were implemented to support FAT32 media. These additions are effective for Windows 95 OEM Service Release 2 and later.

The new Interrupt 21h Function 730X functions can be called using the **DeviceIoControl** function. **DeviceIoControl** has been updated to support VWIN32_DIOC_DOS_DRIVEINFO, a new control code for Interrupt 21h Function 730X functions.

For more information, see **Using VWIN32 to Carry Out MS-DOS Functions**.

The following new interrupt 21h functions have been implemented:

Int21h Function 7302h Get_ExtDPB (FAT32)
Int21h Function 7303h Get_ExtFreeSpace (FAT32)
Int21h Function 7304h Set_DPBForFormat (FAT32)
SetDPB_SetAllocInfo (FAT32)
SetDPB_SetDPBFrmBPB (FAT32)
SetDPB_ForceMediaChng (FAT32)
SetDPB_GetSetActFATandMirr (FAT32)
SetDPB_GetSetRootDirClus (FAT32)
Int21h Function 7305h Ext_ABSDiskReadWrite (FAT32)

Updated IOCTL Functions

All IOCTL functions have been modified to accept a new device category value (CH = 48h) for FAT32 drives. These changes are effective for Windows 95 OEM Service Release 2 and later.

Most of the 08h and 48h forms of each function accept identical parameters. In some cases, the two forms of the call (48h and 08h) are interchangeable. However, this is not always true. As a result, applications that call the 48h form of the function must be prepared to fall back on the 08h form, in case of failure. Please refer to each specific function for information regarding these values.

Set Device Parameters	Int 21h Function 440Dh Minor Code 40h (FAT32)
Write Track on Logical Drive	Int 21h Function 440Dh Minor Code 41h (FAT32)
Format Track on Logical Drive	Int 21h Function 440Dh Minor Code 42h (FAT32)
Set Media ID	Int 21h Function 440Dh Minor Code 46h (FAT32)
Set Access Flag	Int 21h Function 440Dh Minor Code 47h (FAT32)
Lock/Unlock Removable Media	Int 21h Function 440Dh Minor Code 48h (FAT32)
Eject Removable Media	Int 21h Function 440Dh Minor Code 49h (FAT32)
Lock Logical Volume	Int 21h Function 440Dh Minor Code 4Ah (FAT32)
Lock Physical Volume	Int 21h Function 440Dh Minor Code

	4Bh (FAT32)
Get Device Parameters	Int 21h Function 440Dh Minor Code 60h (FAT32)
Read Track on Logical Drive	Int 21h Function 440Dh Minor Code 61h (FAT32)
Verify Track on Logical Drive	Int 21h Function 440Dh Minor Code 62h (FAT32)
Get Media ID	Int 21h Function 440Dh Minor Code 66h (FAT32)
Get Access Flag	Int 21h Function 440Dh Minor Code 67h (FAT32)
Sense Media Type	Int 21h Function 440Dh Minor Code 68h (FAT32)
Unlock Logical Volume	Int 21h Function 440Dh Minor Code 6Ah (FAT32)
Unlock Physical Volume	Int 21h Function 440Dh Minor Code 6Bh (FAT32)
Get Lock Flag State	Int 21h Function 440Dh Minor Code 6Ch (FAT32)
Enumerate Open Files	Int 21h Function 440Dh Minor Code 6Dh (FAT32)
Find Swap File	Int 21h Function 440Dh Minor Code 6Eh (FAT32)
Get Drive Map Info	Int 21h Function 440Dh Minor Code 6Fh (FAT32)
Get Lock Level	Int 21h Function 440Dh Minor Code 70h (FAT32)
Get First Cluster	Int 21h Function 440Dh Minor Code 71h (FAT32)

FAT32 File System Structures

With the addition of the FAT32 file system, the BPB, DPB, and DEVICEPARAMS structures were updated to accommodate FAT32 information. Additionally, subordinate structures have been implemented to support the FAT32 file system main structures. These changes are effective for Windows OEM Service Release 2 and later.

BPB (FAT32)
BIGFATBOOTFSINFO (FAT32)
DPB (FAT32)
EA_DEVICEPARAMETERS (FAT32)
ExtGetDskFreSpcStruc (FAT32)
s_partition (FAT32)
SDPDFormatStruc (FAT32)

Note: Many data structures in this document are listed as **reserved**. This indicates that the user is not to assume or modify the values within these fields. They are used by the file system itself and are not available for any enhancements.

Long Filenames

Microsoft® Windows® 95 allows users and applications to create and use long names for their files and directories. A long filename is a name for a file or directory that exceeds the standard 8.3 filename format.

About Long Filenames

In the past, long filenames typically appeared on network servers that used file systems other than the Microsoft® MS-DOS® file allocation table (FAT) file system. In Windows 95, however, long filenames are available for use with network servers and with local disk drives supporting the protected-mode FAT file system.

This article describes the long filename functions and explains how to create and use long filenames in applications written for MS-DOS and 16-bit Windows version 3. *x*. Microsoft® Win32®-based applications automatically have access to long filenames through the use of the corresponding Win32 file management functions.

Long Filenames and the Protected-Mode FAT File System

The protected-mode FAT file system is the default file system used by Windows 95 for mass storage devices, such as hard disk and floppy disk drives. Protected-mode FAT is compatible with the MS-DOS FAT file system, using file allocation tables and directory entries to store information about the contents of a disk drive. Protected-mode FAT also supports long filenames, storing these names as well as the date and time that the file was created and the date that the file was last accessed in the FAT file system structures.

The protected-mode FAT file system allows filenames of up to 256 characters, including the terminating null character. In this regard, it is similar to the Microsoft® Windows NT® file system (NTFS), which allows filenames of up to 256 characters. Protected-mode FAT allows directory paths (excluding the filename) of up to 246 characters, including the drive letter, colon, and leading backslash. This limit of 246 allows for the addition of a filename in the standard 8.3 format with the terminating null character. The maximum number of characters in a full path, including the drive letter, colon, leading backslash, filename, and terminating null character, is 260.

When an application creates a file or directory that has a long filename, the system automatically generates a corresponding alias for that file or directory using the standard 8.3 format. The characters used in the alias are the same characters that are available for use in MS-DOS file and directory names. Valid characters for the alias are any combination of letters, digits, or characters with ASCII codes greater than 127, the space character (ASCII 20h), as well as any of the following special characters.

$ % ' - _ @ ~ ` ! ( ) { } ^ # &

The space character has been available to applications for filenames and directory names through the functions in current and earlier versions of MS-DOS. However, many applications do not recognize the space character as a valid character, and the system does not use the space character when it generates an alias for a long filename. MS-DOS does not distinguish between uppercase and lowercase letters in filenames and directory names, and this is also true for aliases.

The set of valid characters for long filenames includes all the characters that are valid for an alias as well as the following additional characters.

+ , ; = [ ]

Windows 95 preserves the case of the letters used in long filenames. However, the protected-mode FAT file system, which is not case sensitive, will not allow more than one file to have the same name except for case in the same directory. For example, files named *Long File Name* and *long file name* are not allowed to exist in the same directory. Although extended ASCII characters (characters with ASCII codes greater than 127) are also permitted in filenames, programs should avoid them, because the meanings of the extended characters may vary according to code page. On disk, the characters in the alias are stored using the OEM character set of the current code page, and the long filename is stored using Unicode format.

Although the protected-mode FAT file system is the default file system in Windows 95, it is not the only file system accessible to applications running with Windows 95. For example, applications that connect to network drives may encounter other file systems, such as NTFS. Before using long filenames for files and directories on a volume in a given drive, you must determine the maximum lengths of filenames and paths by using Get Volume Information (Interrupt 21h Function 71A0h function returns values that you can use to make sure your filenames and paths are within the limits of the file system.

In general, you should avoid using static buffers for filenames and paths. Instead, you should use the values returned by Get Volume Information to allocate buffers as you need them. If you must use static buffers, you should reserve 256 characters for filenames and 260 characters for paths. These are the maximum sizes currently recommended for Win32-based applications.

Filename Aliases

When an application creates a file or directory that has a long filename, the system automatically generates a corresponding short filename (alias) for that file or directory, using the standard 8.3 format. Aliases ensure that existing applications that do not handle long filenames can, nevertheless, access those files and directories.

If the long filename follows the standard 8.3 format, the alias has the same name except that all lowercase letters are converted to uppercase. For example, if the long filename is *Examples.Txt*, the corresponding alias will be *EXAMPLES.TXT*.

If the long filename does not follow the standard 8.3 format, the system automatically generates an alias, using the following scheme to ensure that the alias has a unique name. The system tries to create a name by using the first 6 characters of the long filename followed by a *numeric tail*. A numeric tail consists of the tilde (~) character followed by a number. The system starts with the number 1 in the numeric tail. If that filename already exists, it uses the number 2. It continues in this fashion until a unique name is found. If the long filename has a filename extension, the system will use the first three characters of the long filename's extension as the extension for the alias.

As the number of digits in the numeric tail grows, fewer characters in the long filename are used for the 8 characters in the alias. For example, the alias for *Long File Name.File* would be *LONGFI~10.FIL* if the names *LONGFI~1.FIL* through *LONGFI~9.FIL* already existed in the directory. Applications can override the default alias numbering scheme when creating a file by specifying the OPEN_FLAGS_ALIAS_HINT value and supplying a number to use in the call to Create or Open File (Interrupt 21h Function 716Ch).

A period is just another character in a long filename. Leading periods are allowed in a long filename, but trailing periods are stripped. A file can have multiple periods as part of its name. For example, *MyFile.081293.Document* is a valid filename, and its alias will be *MYFILE~1.DOC*. The first three characters after the last period in the filename are used as the filename extension for the alias, as long as the last period is not a leading period. A filename of *.login* is also valid, and its alias is *LOGIN~1*.

In a given directory, the long filename and its alias must uniquely identify a file. For example, if there is a file with the long filename *Long File Name* and the alias *LONGFI~1*, the system will not allow either *Long File Name* or *LONGFI~1* to be used as another file's long filename.

If a file with a long filename is copied or edited, the alias for the resulting file may be different from the original alias. For example, if the destination directory contains an alias that conflicts with the original alias, the system generates another unique alias. If a long filename *LongFileName* is associated with the alias *LONGFI~2* and is later copied to a different directory using the long filename, the alias in the destination directory might be *LONGFI~1* (unless a file with that name already existed in the destination directory). The system always generates new aliases during these operations and always chooses aliases that do not conflict with existing filenames. An application must never rely on an alias being the same for all copies and versions of a given file.

Applications can open, read, and write from a file using the alias without affecting the long filename. However, some operations on the alias, such as copy, move, backup, and restore, may result in the original long filename being destroyed. For example, older versions of utilities that do not support long filenames can destroy the long filename while performing those operations.

The system attempts to preserve a long filename, even when the file associated with it is edited by an application that is not aware of long filenames. Typically, these applications operate on a temporary copy of the file, and when the user elects to save the file, the application deletes the destination file or renames it to another name. The application then renames the temporary file to the destination name or creates a new file with new contents.

When an application makes a system call to delete or rename an alias, the system first gathers and saves

a packet of information about the file and then performs the delete or rename operation. The information saved includes the long filename as well as the creation date and time, the last modification date and time, and the last access date of the original file. After the system performs the delete or rename operation, the system watches for a short period of time (the default is 15 seconds) to see if a call is made to create or rename a file with the same name. If the system detects a create or rename operation of a recently deleted alias, it applies the packet of information that it had saved to the new file, thus preserving the long filename.

Currently, Load and Execute Program (Interrupt 21h Function 4B00h) does not accept long filenames. If an application starts other applications, it must retrieve the filename alias for the given executable file and pass that alias to Load and Execute Program.

File and Directory Management

The standard MS-DOS file and directory management functions do not accept long filenames. You must, therefore, use the long filename functions to create and manage files and directories having long names. The long filename functions are similar to existing MS-DOS system functions. You copy function parameters to registers and issue an Interrupt 21h instruction to carry out the call. The function sets or clears the carry flag to indicate whether the operation was successful and may also return information in registers.

If a long filename function has a corresponding MS-DOS function, the number that identifies the long filename function is four digits long, beginning with the number 71 and ending in the same number as the corresponding MS-DOS function. For example, the long filename function Make Directory (Interrupt 21h Function 7139h) corresponds to MS-DOS Create Directory (Interrupt 21h Function 39h).

You can create or open a file having a long filename by using Create or Open File (Interrupt 21h Function 716Ch). This function takes the name and attributes of the file to create or open and returns a handle that you use to identify the file in subsequent calls to standard MS-DOS functions, such as Read File or Device (Interrupt 21h Function 3Fh) and Write File or Device (Interrupt 21h Function 40h).

You can set or retrieve the time and attributes for a file having a long filename by using Get or Set File Time (Interrupt 21h Function 57h) and Get or Set File Attributes (Interrupt 21h Function 7143h). You can move a file having a long filename by using Rename File (Interrupt 21h Function 7156h) or delete the file by using Delete File (Interrupt 21h Function 7141h).

You can create a directory having a long filename by using Make Directory (Interrupt 21h Function 7139h) or remove the directory by using Remove Directory (Interrupt 21h Function 713Ah).

You can set and retrieve the current directory by using Change Directory (Interrupt 21h Function 713Bh) and Get Current Directory (Interrupt 21h Function 7147h).

File Searches

You can search directories for selected files by using Find First File and Find Next File (Interrupt 21h Functions 714Eh and 714Fh). These functions search for and return information about files having long filenames and aliases (filenames in the standard 8.3 format). The functions return information in a **WIN32_FIND_DATA** structure, which contains both the filename and the corresponding alias, if any.

Unlike MS-DOS Find First File (Interrupt 21h Function 4Eh), the long filename version of Find First File allocates internal storage for the search operations and returns a handle that identifies the storage. This handle is used with Find Next File. To make sure the internal storage is freed, you must use Find Close (Interrupt 21h Function 71A1h) to end the search.

You pass the Delete File (Interrupt 21h Function 7141h) and Find First File functions a filename, which may contain wildcard characters, such as an asterisk (*) or question mark (?). Because Find First File, Find Next File, and Delete File examine long filenames and aliases during the search, some wildcard searches may yield unexpected results. For example, if the system has generated the alias *LONGFI~1* for the long filename *LongFileName*, a search for names that match the **1* pattern would always return the *LongFileName* file, even though that name does not end with a *1*. Searches are not case-sensitive. For example, a search for names that match the *mid* pattern will yield the same results as that for the *MID* pattern. In general, you should check both names returned in the **WIN32_FIND_DATA** structure to determine which of them matched the pattern.

Wildcard searches are more flexible in Windows 95 than in MS-DOS. In the preceding examples, **1* finds the filenames that end in a *1* and **mid** finds filenames that contain the characters *mid*. In MS-DOS and in Windows 95 searching on real-mode FAT directories, all characters after the first * are ignored.

Down-Level Systems

Long filenames, file last access date, and file creation date and time are not supported while the file system is in single MS-DOS application mode. They are not supported either in versions of MS-DOS that only use the real-mode FAT file system. These file systems and others that do not support long filenames are referred to as down-level systems. If you intend for an application to run with both Windows 95 and down-level systems, you should always check the system to determine whether it supports the long filename functions. The easiest way to check is to call Get Volume Information (Interrupt 21h Function 71A0h). This function returns an error if the system does not support the long filename functions.

Another way of handling down-level systems is to use a combination of calls to long filename and standard MS-DOS functions to carry out file management. In this case, you call the standard function only if the long filename function is not supported. To indicate an unsupported function, the system sets the AL register to zero but leaves the AH register and the carry flag unchanged. The following example shows how to combine long filename and standard functions to carry out a file or directory management operation.

```
stc             ; set carry for error flag
                ; set registers here for LFN function call
int 21h         ; call long filename function
jnc success     ; call succeeded, continue processing
cmp ax, 7100h   ; is call really not supported?
jne failure     ; supported, but error occurred
                ; set registers here for MS-DOS function call
int 21h         ; call standard MS-DOS function
```

Application developers have to decide what to do when users save a file with a long filename to a down-level system. One approach is to imitate the behavior of the command interpreter (COMMAND.COM) and save the file using the alias without informing the user. A different approach is to have the application inform the user that the file system does not support long filenames and allow the user to save the file with a filename in the standard 8.3 format.

Last Access Date

The Windows 95 last access date is intended to reflect the last time a file was accessed for the purpose for which it was created. This date is intended to provide a means for applications, users, or both to determine which files have not been used recently. When an application saves a file, the system automatically resets the last access date. An application that cannot understand the contents of the files it is accessing should save the last access date and restore it after closing the file. For example, applications that back up a file, search files for strings, and scan for viruses should save and restore the last access date.

Applications should allow the system to set the last access date in the following cases:

- Running a program should set the last access date for the .EXE file.
- Loading a dynamic-link library (DLL) should set the last access date for the .DLL file.
- Editing or printing a document should set the last access date for the document file.
- In general, any use of a document by an application that creates or modifies that type of document should set the last access date (unless the document is being opened only to decide whether it is to be used in a find operation).
- Application use of peripheral files (.INI files and so on) should set the last access date.

Win32-based applications can preserve the last access date by using the **GetFileTime** and **SetFileTime** functions. Applications written for MS-DOS or Windows version 3. *x* can use Get Last Access Date and Time (Interrupt 21h Function 5704h) and Set Last Access Date and Time (Interrupt 21h Function 5705h), or they can open the file with Create or Open File (Interrupt 21h Function 716Ch) using the OPEN_ACCESS_RO_NOMODLASTACCESS (0004h) access mode.

Long Filenames Reference

The long filename functions match the following Win32 file management functions.

Long filename function	Win32 function
Interrupt 21h Function 5704h Get Last Access Date and Time	**GetFileTime**
Interrupt 21h Function 5705h Set Last Access Date and Time	**SetFileTime**
Interrupt 21h Function 5706h Get Creation Date and Time	**GetFileTime**
Interrupt 21h Function 5707h Set Creation Date and Time	**SetFileTime**
Interrupt 21h Function 7139h Make Directory	**CreateDirectory**
Interrupt 21h Function 713Ah Remove Directory	**RemoveDirectory**
Interrupt 21h Function 713Bh Change Directory	**SetCurrentDirectory**
Interrupt 21h Function 7141h Delete File	**DeleteFile**
Interrupt 21h Function 7143h Get or Set File Attributes	**GetFileAttributes**, **SetFileAttributes**
Interrupt 21h Function 7147h Get Current Directory	**GetCurrentDirectory**
Interrupt 21h Function 714Eh Find First File	**FindFirstFile**
Interrupt 21h Function 714Fh Find Next File	**FindNextFile**
Interrupt 21h Function 7156h Rename File	**MoveFile**
Interrupt 21h Function 7160h Minor Code 0h Get Full Path Name	**GetFullPathName**
Interrupt 21h Function 7160h Minor Code 1h Get Short Path Name	**GetShortPathName**
Interrupt 21h Function 7160h Minor Code 2h Get Long Path Name	No Win32 function equivalent
Interrupt 21h Function 716Ch Create or Open File	**CreateFile**, **OpenFile**
Interrupt 21h Function 71A0h Get Volume Information	**GetVolumeInformation**
Interrupt 21h Function 71A1h Find Close	**FindClose**
Interrupt 21h Function 71A6h Get File Info By Handle	**GetFileInformationByHandle**
Interrupt 21h Function 71A7h Minor Code 0h	**FileTimeToDOSDateTime**

File Time To DOS Time	
Interrupt 21h Function 71A7h Minor Code 1h	**DOSDateTimeToFileTime**
DOS Time To File Time	
Interrupt 21h Function 71A8h	No Win32 function equivalent
Generate Short Name	
Interrupt 21h Function 71A9h	No Win32 function equivalent
Server Create or Open File	
Interrupt 21h Function 71AAh Minor Code 0h	No Win32 function equivalent
Create Subst	
Interrupt 21h Function 71AAh Minor Code 1h	No Win32 function equivalent
Terminate Subst	
Interrupt 21h Function 71AAh Minor Code 2h	No Win32 function equivalent
Query Subst	

Note that Interrupt 21h Functions 71A2h through 71A5h exist, but they are for internal use by Windows 95 only.

MS-DOS Extensions

This article describes various MS-DOS extensions for Windows 95 and provides general information about the file system.

About MS-DOS Extensions

Microsoft® Windows 95® not only supports the complete set of Microsoft® MS-DOS® system functions and interrupts but also provides extensions that permit MS-DOS - based applications to take advantage of Windows 95 features, such as long filenames, exclusive volume locking, virtual machine services, message services, and program information file management. For more information about these topics, see the articles with those names in this guide. For more information about MS-DOS functions and interrupts, see the *Microsoft MS-DOS Programmer's Reference*.

Windows 95 Version of MS-DOS

When running with Windows 95, MS-DOS - based applications can check the operating system version by using Get MS-DOS Version Number (Interrupt 21h, Function 30h). For Windows 95, this function returns 7 as the major version number and 0 as the minor version number.

If you need to know that the system is running MS-DOS version 7.0, you must use Interrupt 2Fh Function 4A33h. This function returns zero in the AX register for MS-DOS version 7.0 or higher and returns a nonzero value in AX for any other versions of the disk operating system. In addition to the AX register, this function uses the DS, SI, DX, and BX registers.

File System Support

Windows 95 supports the long filename file allocation table (FAT) when running Windows. Any physical file on this extended FAT file system will logically be associated with two names – namely, the *primary filename* (also referred to as the long filename) and its *alternate name or alias*. Windows 95 automatically generates the alias, and it is always in the standard 8.3 filename format. When a file is saved to disk, the system creates a directory entry for both the long filename and alias. Because the number of entries in the root directory is limited, it is best to store files in a directory below it to avoid filling up the root. For more information about the filename conventions and how the system generates the alias, see Long Filenames.

Filename Limitations Under Real Mode

If Windows 95 is started in single MS-DOS application mode (real mode), only the standard FAT file system (and not the long filename FAT file system) is supported. This means that long filenames that are created in a Windows environment will *not* be visible when the user exits to single MS-DOS application mode, although the names themselves are physically present on the media. Only the alias (the 8.3 filename) will be visible.

When down-level file systems (such as MS-DOS version 6.0, Windows version 3.1, Microsoft® Windows NT® version 3.1, and OS/2® version 2.11) read a floppy disk that contains long filenames created using Windows 95, the long filename will not be visible; only the alias (the 8.3 filename) will be visible. However, Windows NT version 3.5 supports long filenames. Windows 95 will see the long filenames of files on a floppy disk that were created using Microsoft Windows NT version 3.5, and Windows NT version 3.5 will see the long filenames of files on a floppy disk that were created using Windows 95. Windows 95 will be able to see the long filenames on New Technology file system (NTFS) or Novell NetWare's file system if there are long filenames on the server.

Because down-level systems are not aware of long filenames, they will not preserve them. If you copy a file from a floppy disk to the hard disk on a down-level system, the long filename associated with the file is not copied over. If you edit a file on the floppy disk using the alias and then save a new copy back on the floppy disk using the down-level system, the long filename associated with the file will most likely be lost. If you take the floppy disk back to the Windows 95 system, only the alias will be associated with the file.

Preserving Filenames

Certain operations, such as copy, backup, and restore, using older versions of utilities that have not been updated to support long filenames will destroy the long filename. Running a utility called LFNBK that comes with Windows 95 before using older backup utilities will preserve the long filenames. Following are the steps for using LFNBK to backup and restore a disk:

1. Preserve the long filename and alias (the 8.3 filename) association by running **LFNBK /b** *drive-letter*, where *drive-letter* is the drive that you plan to back up.
2. Back up the drive (specified as *drive-letter* in the previous step) using an old backup program that is not be aware of long filenames.
3. Restore the backup files to a drive, when necessary at some later point, using an old restore program that is not be aware of long filenames.
4. Restore the long filenames on the drive, by running **LFNBK /r** *drive-letter*, where *drive-letter* is the drive where the files were restored to.

Searching Filenames

Searches of filenames apply to both the filename and its alias. The system presents a *single unified namespace* so that a single physical view of the file is preserved. However, if the result of a search shows only the long filename, it could be confusing to the user. For example, a set of files in a directory might include the following filenames and aliases.

Filename	**Alias**
LongFileName	LONGFI~1
File-1	FILE-1

A search of files in the directory using **DIR *1** would display the following information.

```
LONGFI~1     123     05-11-95     15:26     LongFileName
FILE-1       352     05-11-95     16:01     File-1
```

Note that the **DIR** command displays the alias first for compatibility with the older **DIR** format. However, a search utility that is aware of long filenames but displays only the filename, would also show both the *LongFileName* and *File-1* files. This could be confusing at first glance to the user because the *LongFileName* file does not have the number *1* as specified in the search criteria. The file was matched to the search pattern because its alias contains the number *1*.

The wildcard searches have been expanded in Windows 95. Using the old search criteria, the first ***** encountered caused all following characters to be ignored. However, in the preceding example, ***1** is a valid specification. More than one wildcard can be used in Windows 95 when specifying search criteria. For example, to search for all files that contain the word **mid** somewhere in the filename, ***mid*** can be specified as the search criteria.

Performing Direct Disk Writes

Previously reserved fields in the file system directory entry are used for storing the last access date and the creation date and time. A file with a long filename, which is longer than 5 characters, will also cause the system to use previously reserved fields in the directory entry. In previous versions of MS-DOS and Windows, these reserved entries were zero. Older disk repair and checking utilities that have not been updated for Windows 95 might display errors about the disk because of the usage of these reserved entries. It is possible that an older disk repair utility could either destroy the long filename or the actual data in a file because it would mistakenly interpret the file as corrupted. For this reason, Windows 95 won't support old utilities that perform direct disk writes. A newer version of the utility that understands the newer on-disk file system structures will obtain an exclusive volume lock and proceed correctly. This feature is called exclusive volume locking.

Exclusive volume locking is also needed because the system is a multitasking system and disk utilities need exclusive access so that they can modify the file system without causing the file system to be inconsistent for the other executing applications.

Storing Filenames

When an application stores filenames, it should follow these guidelines:

- Using an alias instead of a long filename can break the association with the file because certain operations, such as editing the file, can potentially change the alias. For example, if the server side of a network uses the alias instead of the long filename of a client file to store information related to the file (such as access permissions), it will be more susceptible to lose the association of permissions with the file. If the user edits the file on the client side, the alias may potentially change, and if it changes, the server side will lose the associated information.
- When an application caches the absolute path to a filename, the path itself can be a mixture of filenames and aliases. Because of this, applications that need to store a path should store a canonical form of the absolute path. Applications may use either Get Short Path Name (Interrupt 21h Function 7160h Minor Code 1h) or te **GetShortPathName** function to retrieve the canonical form of a path. Applications that need to determine if two files are the same can use the **nFileIndexHigh** and **nFileIndexLow** members of the **BY_HANDLE_FILE_INFORMATION** structure. Note, however, that **nFileIndexHigh** and **nFileIndexLow** might not be supported on real-mode file systems, such as Microsoft CD-ROM Extensions (MSCDEX), or real-mode networks.
- An installation program that needs to enter information into configuration files, such as CONFIG.SYS, should make sure it uses paths that only consist of 8.3 filename components, because the long filenames will not be visible at boot up time when startup files, such as CONFIG.SYS and AUTOEXEC.BAT, are processed. Again, the **GetShortPathName** function should be used for this purpose.

Filename Functions

With the exception of the **OpenFile** function, all of the functions in Windows version 3. *x* that require the application to pass in a filename (such as **LoadLibrary**, **WinExec**, **_lopen**, and **_lcreate**) have been updated to support long filenames. For compatibility reasons, functions that return filenames should return only aliases (8.3 filenames) to 16-bit Windows - based applications marked less than 4.0.

MS-DOS - based applications generally use the Interrupt 21h functions. Except for Extended Open/Create (Interrupt 21h Function 6Ch), the older Interrupt 21h functions have not changed in Windows 95. Extended Open/Create has been enhanced in Windows 95 to make use of the last access date for a file. To support long filenames, Windows 95 provides many new Interrupt 21h functions. Any MS-DOS - based application that will use long filenames must be updated to support the new functions. For information about the long filename Interrupt 21h functions, see Long Filenames

Windows 95 supports the new Interrupt 21h long filename functions on as many file systems as possible. On file systems, such as MSCDEX, Flash, and real-mode network redirectors, that do not support long filenames, the system automatically translates the newer Interrupt 21h calls to the appropriate older Interrupt 21h calls, as long as the filename passed as a parameter is a valid alias (8.3 filename). Applications may use Get Volume Information (Interrupt 21h Function 71A0h) to retrieve information on the capabilities of the underlying file system.

Command Interpreter for Command

The **for** command in the command interpreter (COMMAND.COM) is modal. The default is **LFNFOR=OFF**, which causes the **for** command to use the old Interrupt 21h function calls. In that case, only aliases (8.3 filenames) can be used in the **for** command. If **LFNFOR=ON** is set, the **for** command uses the new Interrupt 21h functions, and long filenames can be used as part of the **for** command.

Long Command Lines

Although in previous versions of MS-DOS the limit for environment variables and batch file lines is 128 characters, it is 1024 characters in Windows 95. The limit for the keyboard buffer, however, is still 128 characters. Although 1024 and 128 are the standard limits, users may configure their systems to lower these limits.

In previous versions of MS-DOS, command-line arguments are located in the command tail of the program segment prefix (PSP). The command tail in the PSP is limited to 128 characters, including the leading byte that specifies the length of the command line and the trailing carriage return character. In Windows 95, if the command line is less than or equal to 126 characters, it is set in the command tail of the PSP. For command lines that are greater than 126 characters, an application should follow these steps:

1. Set the count byte in the command tail to 7Fh.
2. Fill in 7Eh bytes of the command tail followed by the carriage return character (0Dh).
3. Place the rest of the command line in the CMDLINE environment variable.

MS-DOS Extensions Reference

This section provides information about the following functions and structures.

Interrupt 21h Function 4302h	Get compressed file size.
Interrupt 21h Function 440Dh Minor Code 48h	Lock/unlock removable media.
Interrupt 21h Function 440Dh Minor Code 49h	Eject removable media.
Interrupt 21h Function 440Dh Minor Code 6Fh	Get drive map information.
Interrupt 21h Function 440Dh Minor Code 71h	Get first cluster.
Interrupt 21h Function 6Ch	Extended open/create.
DRIVE_MAP_INFO	Get Drive Map Info returns information about the specified drive in this structure.
MID	This structure, which is used by Get Media ID (Interrupt 21h Function 440Dh Minor Code 66h) and Set Media ID (Interrupt 21h Function 440Dh Minor Code 46h), has been updated to support compact disc (CD) file systems. For information about using these functions, see the *Microsoft MS-DOS Programmer's Reference.*
PARAMBLOCK	This structure is needed by Lock/Unlock Removable Media.

Thunk Compiler

These topics describe how to use the thunk compiler to avoid porting all of your code to 32 bits or to take advantage of 16-bit components in order to save development time. In addition, these topics describe issues related to the thunk compiler that may make you choose to avoid it in your application.

About the Thunk Compiler

Writing applications for the Microsoft® Win32® application programming interface (API) is recommended for these reasons:

- The flat address space – that is, no segments or offsets
- Better performance
- Greater robustness

However, as compelling as these arguments are, there may be pragmatic reasons for not making the move all at once. Certain components of an application may lend themselves to the migration to 32 bits, while other components may be more tightly bound to the 16-bit environment. For example, an application developer may want to move the application's user interface code to 32 bits to take advantage of new system features, but may have an existing dynamic-link library (DLL) specifically optimized for the 16-bit architecture. Rather than delay the release of a new version of the application, the developer can decide instead to take advantage of the thunk compiler to mix 16- and 32-bit components on Windows 95.

Thunking Mechanics

The thunk compiler needs to check for these elements in a mixed 16- and 32-bit environment:

- Pointers consist of a selector and a 16-bit offset in a 16-bit Windows environment. However, in a 32-bit Windows environment, pointers essentially consist of a 32-bit offset; that is, the DS, ES, SS, and CS registers all contain selectors with the same base address. All pointers in this environment are considered to be near 0:32 pointers. Translating a 16:16 pointer to a 0:32 pointer involves determining the segment base for the selector portion of the pointer and adding the offset to it. Translating a 0:32 pointer to a 16:16 pointer involves allocating a selector and calculating the offset from the base of the corresponding segment.
- Both 16- and 32-bit applications pass function parameters on the stack. However, 16-bit applications address the stack using the SS:SP registers, but 32-bit applications use the SS:ESP registers. When thunking from 16 bits to 32 bits (or thunking back in the other direction), the processor's method of addressing the stack must be switched.
- The word size is 32 bits in a 32-bit environment and 16 bits in a 16-bit environment. When an application uses both 16- and 32-bit environments, some piece of software must be able to translate 16-bit words to 32-bit words. The following 16-bit function can be used to illustrate the translation.

  ```
  DWORD Sample(WORD i);
  ```

  If a 32-bit component calls this 16-bit function, it pushes a 32-bit argument on the stack, but the 16-bit function only pops 16 bits off the stack. Later, when the function returns, it places the returned doubleword value in the DX:AX register pair, but the 32-bit calling application expects the return value to be in the EAX register. It is the thunking layer's responsibility to negotiate these translations.
- To correctly handle packing and conversion, the thunk compiler translates structure members one at a time. The thunk compiler supports structure packing on 1-, 2-, or 4-byte boundaries (corresponding to the /Zp1, /Zp2, and /Zp4 compiler options for Microsoft C/C++ compilers). However, 8-byte structure alignment is not supported. Structure packing boundaries must be the same on both sides of a thunk.
- Unlike 32-bit code, 16-bit code is usually not reentrant; 16-bit code is typically written with the assumption of a cooperative mulitasking model. The thunk layer must serialize access to 16-bit code. In contrast, 32-bit code must execute without serialization to prevent deadlocks. The thunk layer manages serialization on transitions in both directions.

Thunking Benefits and Drawbacks

If you use thunking to avoid porting all of your code to 32 bits or to take advantage of 16-bit components, you may save development time. In addition, your executable files may have a smaller memory footprint than if they were full 32-bit applications. However, there are some issues related to thunking that may make you choose to avoid it in your application. These issues are described in the following sections.

Thunking Models

Windows NT and Windows 95 use different thunking models. Window NT supports generic thunks, which only allow thunking from 16 bits to 32 bits. Although Windows 95 supports generic thunks, it does not support the underlying process model used by Windows NT. Use of the Windows 95 model can lead to serious incompatibilities with the Windows NT generic thunking model. Windows 95 implements its own thunking model, called "flat thunks." Flat thunks allow thunking from 32 bits to 16 bits or vice versa. The flat thunking model requires use of the thunk compiler.

If you use flat thunks, your application cannot run on Windows NT.

Compatibility with Existing 16-Bit DLLs

The thunk compiler accepts the following compatibility statement.

```
win31compat = true;
```

You must use this statement if a 16-bit DLL replaces a DLL that is currently used in a Windows version 3.1 environment or if the DLL runs as part of a graphics device interface (GDI) device driver. You should probably use this statement even if your 16-bit DLL does not fit into one of these categories, because it provides extra protection against poorly behaved DLLs that might be used by your thunk DLL.

The compatibility statement causes the unloading of a 32-bit DLL to be deferred until the containing process terminates. Use of this statement ensures that interprocess loading and freeing procedures, which may have worked with a pure 16-bit DLL, will not cause the thunked version of the 32-bit DLL to be freed prematurely. It also allows the 16-bit library to be freed without execution of the 32-bit DLL's notification routine – an occurrence that could cause reentry into 16-bit code.

16- to 32-Bit Thunks and Preemption

Because 16-bit processes multitask cooperatively with respect to each other, 16-bit code is often written with the assumption that it will not be interrupted by another process until it explicitly yields the 16-bit scheduler (typically by calling the **GetMessage** function). In contrast, 32-bit code is written with the expectation that it can wait on synchronization objects without impeding the progress of other processes. If you mix code written under these different assumptions, you should take care to avoid deadlocks or errors due to unexpected reentrancy.

While a process is executing 32-bit code, it is possible for other processes to enter 16-bit code, possibly reentering the 16-bit code that was thunked from. That is, entering a 16- to 32-bit thunk releases the 16-bit subsystem for use by other processes (under certain conditions noted in the following paragraph). Thus, 16- to 32-bit thunks should not be used if your code cannot be reentered at that time. In addition, you should avoid using 16- to 32-bit thunks inside callback functions passed to a third-party DLL, unless it is documented that the code calling the callback function is reentrant. For example, if the callback function is allowed to yield, it is probably reentrant.

If the process executing the thunk is 32 bits, any other process can reenter 16-bit code inside a 16- to 32-bit thunk. On the other hand, if the process executing the thunk is 16 bits, only 32-bit processes can reenter 16-bit code, because 16-bit processes are cooperatively multitasked with respect to each other. Thus, if your 16-bit code is used only by 16-bit processes, the reentrancy requirement can be relaxed, because other 16-bit processes can get control only if your 16-bit process yields. (A DLL cannot control what applications call it, of course, so guaranteeing that your 16-bit code will be used only by 16-bit processes is difficult.) There are other restrictions on what can be done in 16-bit processes, as described in the next section.

Behavior of Win32 Functions Inside 16-Bit Processes

In Windows 95, 16- to 32-bit thunks are primarily useful for implementing callback thunks on 32-bit processes. They can also be used to execute 32-bit code in 16-bit processes. Although the latter can improve performance and multitasking, there are important restrictions on what can be done in a 16-bit process.

A 16-bit process executing 32-bit code is not the same as a 32-bit process executing the same code. The following important differences apply:

- The 32-bit code will still execute on the stack reserved by the 16-bit application, which is much smaller than the 1MB stack used by true 32-bit applications.
- 16-bit processes cannot create new threads. Certain Win32 API elements, such as the functions supporting the new common dialog boxes or those supporting console applications, create threads on behalf of the calling application. These functions cannot be used in a 16-bit process.
- Thunking into 32-bit code releases the 16-bit subsystem to other 32-bit processes, but other 16-bit processes will still be blocked, unless your process explicitly yields (typically by calling the **GetMessage** function). In other words, 16-bit applications still multitask cooperatively even when executing 32-bit code. This means that no 16-bit application will get processing time if a 16-bit application blocks for a long time without yielding. A common problem is using the **CreateProcess** function to launch a 16-bit application and then waiting on a synchronization object to do something. If the application does not yield, the new application will never run and signal the object, so deadlock will occur.

  One way to avoid this problem is to use the **MsgWaitForMultipleObjects** function to wait for either messages or synchronization objects. Although this solution is effective when required, it still results in a less efficient blocking operation than that of a 32-bit application.

In general, 32-bit code within 16-bit processes should be limited to code that uses the 32-bit heap functions, memory-mapped file functions, file functions, and functions involving the current process and thread. 32-bit code using GDI, dialog box, message box, and message functions will also work within 16-bit processes. You should avoid using third party libraries, unless you are sure they work safely in a 16 bit-environment.

Globally Fixing Handles

Translating a 16:16 pointer to a 0:32 pointer involves determining the segment base for the selector portion of the pointer and adding the offset to it. The global memory compacter in Windows 95 may move a block of memory at any time, however, making the linear address invalid. When the thunk compiler converts pointers from 16 bits to 32 bits, it fixes the segment portion before computing the linear address. If you translate pointers without using the thunk compiler, however, you must be aware of the requirement to fix the segment portion. For more information about the functions you can use to translate pointers, see Translating 16:16 Pointers .

16- to 32-Bit Thunks in GDI Device Drivers

The following compatibility statement is required for any 16- to 32-bit thunk script that runs as part of a GDI device driver.

```
win31compat = true;
```

You may only thunk using the following control display driver interface (DDI) functions in a GDI device driver.

Control(ABORTDOC)
Control(ENDDOC)
Control(NEWFRAME)
Control(NEXTBAND)
Control(STARTDOC)

If you use thunking, your driver must be reentrant.

You may call the **OpenJob**, **StartSpoolPage**, **EndSpoolPage**, **CloseJob**, and **DeleteJob** functions during the five listed control DDI calls. These calls will go through drivers written for Windows version 3.*x* for the sake of backward compatibility, but they will fail for Windows 95-based drivers.

Using the Thunk Compiler

The Microsoft Win32 Software Development Kit (SDK) includes a sample application that illustrates the use of the thunk compiler. This application, APP32.EXE, simply thunks some basic types from 32 bits to 16 bits. This sample is an important supplement to the information in this topic.

The thunk compiler's input is a "thunk script," which is a list of C style function prototypes and type definitions. The compiler produces an .ASM file, which is really two .ASM files in one. You can assemble this .ASM file with the **-DIS_16** option to get the 16-bit .OBJ file to link to the 16-bit component and then assemble it with the **-DIS_32** option to get the 32-bit .OBJ file to link to the 32-bit component.

The 16-bit component of the .ASM file contains a jump table containing the 16:16 address of each function named in the thunk scripts. (The linker must be able to resolve these references; the functions must use the Pascal calling convention and either be implemented in the 16-bit DLL or be imported by the DLL.) The 32-bit half of the .ASM file contains a Stdcall function for each thunk, which converts its parameters to 16 bits and then employs some internal processing to call the 16-bit target referenced in the jump table. When a 32-bit application uses a thunked function, it calls these compiler-generated Stdcall functions directly.

For example, a thunk script's declaration for the **LineTo** function might look like this.

```
typedef          int INT;
typedef unsigned int UINT;
typedef UINT         HANDLE;
typedef HANDLE       HDC;

BOOL LineTo(HDC, INT, INT) {
}
```

For more information about thunk script files, see Script Files.

An application would never include the **LineTo** function in a thunk script, of course, because this function already exists in 16- and 32-bit versions. This example (and the assembly language example that follows) are intended to illustrate the process; the assembly language code, in particular, could differ from the actual code that is generated by the current version of the thunk compiler.

When the preceding example from a thunk script is processed by the thunk compiler, the following assembly language code is generated. On the 16-bit half, there is the following jump table.

```
externDef LineTo:far16

FT_gdiTargetTable label word
    dw    offset LineTo
    dw       seg LineTo
```

The 32-bit half contains the following code.

```
public LineTo@12
LineTo@12:
    mov    cl,0
; LineTo(16) = LineTo(32) {}
;
; dword ptr [ebp+8]:  param1
; dword ptr [ebp+12]:  param2
; dword ptr [ebp+16]:  param3
;
```

```
public IILineTo@12
IILineTo@12:
    call    QT_Entry
    push    word ptr [ebp+8]    ;param1: dword->word
    push    word ptr [ebp+12]    ;param2: dword->word
    push    word ptr [ebp+16]    ;param3: dword->word
    call    QT_Target_gdi
    movsx    ebx,ax
    jmp        QT_Exit12
```

When a Win32-based application calls the **LineTo** function, it transfers directly to this routine, which builds a 16-bit call frame and calls a local routine asking it to look up the appropriate address in the jump table and sign-extend the return value. (Each component receives its own set of QT_ routines, which automatically use the correct jump table. The QT_ and FT_ routines are exported by the kernel.)

Script Files

Script files contain descriptions of the functions that are thunked. These files usually have a .THK filename extension. The script files are easily created using function prototypes. For example, a function might be prototyped in the following manner.

```
BOOL WINAPI Sample(int n);
```

The corresponding definition would look like this in the script file.

```
typedef bool BOOL;
typedef int INT;

BOOL Sample(INT n)
{
}
```

Many functions take pointers in their parameter lists. Some pointers are for input only, some are output only, and some are for both input and output. For example, a "ThunkIt" function might take a pointer to an input string, update a second string, and produce a third string as output in the following manner.

```
BOOL WINAPI ThunkIt(LPSTR lpstrInput, LPSTR lpstrInOut,
    LPSTR lpstrOutput);
```

The corresponding thunk script declaration for the function follows.

```
typedef char *LPSTR;

BOOL ThunkIt(LPSTR lpstrInput, LPSTR lpstrInOut,
    LPSTR lpstrOutput)
{
    lpstrInput = input;      // optional, because pointers are input
                             // by default
    lpstrInOut = inout;      // pointer taken in and updated
    lpstrOutput = output;    // pointer returned
}
```

Note that the thunk compiler does not need WINAPI in the thunk script, because it expects that the functions are declared with WINAPI. Adding WINAPI to the thunk script will cause a syntax error.

When a pointer is passed from 32-bit to 16-bit code, a single selector with a limit of 64K is allocated in the thunk. If the 16-bit code needs to access more than the first 64K of the block, it must change the base address of the selector or allocate additional selectors to access the block.

The thunk compiler supports the following constructions:

- Structures passed by value or reference.
- Structures within structures.
- Pointers within structures, provided that the object pointed to does not require repacking. The object can be another structure.
- Arrays of scalar values embedded in structures.
- The "input", "output," and "inout" qualifiers for pointers, as shown in the preceding example. The default qualifier is "input."
- Returning pointers for 32- to 16-bit thunks, provided that the object pointed to requires no repacking.

The object can be a structure. The segment is not globally fixed by the thunk compiler. As a general rule, the thunk compiler deallocates the selectors that it allocates.

- The **bool** type. This type is preferred over **int** in situations where an application may use nonzero values other than 1 to represent TRUE.

The thunk compiler does not support arrays of pointers, arrays of structures, unions, or floating-point types (such as **float** or **double**). In addition, thunked functions can only support up to 14 parameters.

Procedure for Adding Flat Thunks

You should follow these steps to add flat thunks:

1. Write a thunk script containing thunk declarations and type definitions for the functions that need to be thunked. You should place the following line at the beginning of the script to create 32- to 16-bit thunks.

   ```
   enablemapdirect3216 = true;     // creates 32 to 16 thunks
   ```

   Alternatively, you can use this line to create 16- to 32-bit thunks.

   ```
   enablemapdirect1632 = true;     // creates 16 to 32 thunks
   ```

2. Compile your thunk script.

   ```
   thunk.exe <inputfile> [-o <outputfile>]
   ```

   The thunk compiler has the following command line.

   thunk [{-|/}*options*] *infile*[.*ext*]

?	Displays the help screen.
h	Displays the help screen.
o *name*	Overrides the default output filename.
p *n*	Changes the 16-bit structure alignment (default = 2).
P *n*	Changes the 32-bit structure alignment (default = 4).
t *name*	Overrides the default base name.
N*x name*	Specifies the name segment or class where *x* is either C32 (for 32-bit code segment name) or C16 (for 16-bit code segment name).

3. Assemble the resulting .ASM file to create the 16-bit side of the thunk, and make sure to define the **DIS_16** option. For example, using Microsoft MASM version 6.11, you might use the following command line.

   ```
   ml /DIS_16 /c /W3 /nologo /Fo thk16.obj 32to16.asm
   ```

4. Make sure to mark your 16-bit DLL as being compatible with subsystem version 4.0 by running the Microsoft Resource Compiler (RC.EXE), which is included in the Win32 SDK, on the DLL.

   ```
   \mstools\bin16\rc -40 <.DLL output file>
   ```

Important If the 16-bit DLL is not marked for subsystem version 4.0, the 32-bit DLL will not load. For more information, see Troubleshooting.

5. Assemble the resulting .ASM file to create the 32-bit side of the thunk, and make sure to define the **DIS_32** option. For example, using Microsoft MASM version 6.11, you might use the following command line.

   ```
   ml /DIS_32 /c /W3 /nologo /Fo thk32.obj 32to16.asm
   ```

6. Add the entrypoint functions to the DLLs and the export and import statements to their module-definition (.DEF) files.
7. Compile and link the 16- and 32-bit components.
8. The code generated by the thunk compiler links to several Windows 95 entrypoint functions in

KRNL386.EXE and KERNEL32.DLL. These entrypoint functions are specific to the thunk compiler and are not supported in Windows NT.

The entrypoint functions for KERNEL32.DLL are defined in THUNK32.LIB. You must link your 32-bit DLL with this library file.

In addition, the following two import statements should be added to the .DEF file for the 16-bit DLL.

```
C16ThkSL01     = KERNEL.631
ThunkConnect16 = KERNEL.651
```

Once again, these entrypoint functions exist only in Windows 95 and not in Windows NT. Attempts to load DLLs that use these functions will fail in Windows NT.

Implementing a Thunking Layer

The thunking model allows either the 16- or 32-bit thunk component to start first and clean up everything afterward. Follow these steps to implement the thunking layer:

1. Add a procedure named "DllEntryPoint" to your 16-bit DLL. This procedure should look like the following example. (The names of the DLLs are for illustration only.)

```
BOOL FAR PASCAL __export SAMP_ThunkConnect16(LPSTR pszDll16,
    LPSTR pszDll32, WORD hInst, DWORD dwReason);

BOOL FAR PASCAL __export DllEntryPoint(DWORD dwReason, WORD hInst,
        WORD wDS, WORD wHeapSize, DWORD dwReserved1,
        WORD wReserved2) {
    if (!(SAMP_ThunkConnect16("DLL16.DLL", // name of 16-bit DLL
            "DLL32.DLL",                   // name of 32-bit DLL
            hInst, dwReason))) {
        return FALSE;
    }
    return TRUE;
}
```

   In this example, "SAMP" is the base name – that is, the name of the thunk script file, not including the path and filename extension. If you used the **/t** option with the thunk compiler to specify a different base name, you would use the new base name in your procedure. You could, for example, use following command line.

```
THUNK /t MYNAME SAMP.THK
```

   The thunk compiler generates the MYNAME_ThunkConnect16 routine, which should then be called in the DllEntryPoint procedure.

2. Add the following import and export statements to your 16-bit DLL's module definition (.DEF) file, picking ordinals that are appropriate for your DLL.

```
EXPORTS
DllEntryPoint       @1 RESIDENTNAME
SAMP_ThunkData16    @2
IMPORTS
C16ThkSL01     = KERNEL.631
ThunkConnect16 = KERNEL.651
```

3. Add the following function to your 32-bit DLL's entrypoint procedure (named DllMain by default). Again, the names of the DLLs are for illustration only.

```
BOOL WINAPI SAMP_ThunkConnect32(LPSTR pszDll16, LPSTR pszDll32,
    DWORD hIinst, DWORD dwReason);
BOOL _stdcall DllMain(DWORD hInst, DWORD dwReason,
        DWORD dwReserved) {
    if (!(SAMP_ThunkConnect32("DLL16.DLL", // name of 16-bit DLL
            "DLL32.DLL",                   // name of 32-bit DLL
            hInst, dwReason))) {
        return FALSE;
    }
    // Process dwReason.
}
```

4. Add the following export statement to your 32-bit DLL's .DEF file.

```
EXPORT
SAMP_ThunkData32
```

The following implementation rules are very important:

- The 16-bit **DllEntryPoint** function is called each time the module's usage count is incremented or decremented. The *dwReason* parameter is 1 when the usage count is incremented and zero when it is decremented.
- Because the system calls the **DllEntryPoint** function while loading is underway, the value returned by the **GetModuleUsage** function is not reliable if you call it inside your **DllEntryPoint** function.
- Do not call thunks inside the **DllEntryPoint** routines. Do not perform operations that reenter the 16-bit loader, yield the 16-bit scheduler, or release the 16-bit subsystem. These actions could cause unpredictable results and are not guaranteed to work in future versions of the thunk compiler.
- You can have multiple **ThunkConnect***XX* calls, so long as they connect to the same DLL. In fact, this is the only way to get both 32- to 16-bit and 16- to 32-bit thunks in one thunk-paired DLL set. The procedure is to have a thunk script for each direction and link both into each DLL. Then each entrypoint function will have two **ThunkConnect** calls.

Late Loading

Windows 95 improves system performance by supporting late loading for thunk DLLs. Loading the 16-bit DLL does not cause the corresponding 32-bit DLL to load immediately. Instead, the thunk subsystem loads the 32-bit DLL when the first 16- to 32-bit thunk is started.

Late loading has the following implications:

- The performance and working set are improved for 16-bit applications that use only the 16-bit portions of thunked DLLs. The 32-bit DLL will not load into those processes.
- The 16-bit DLL must not depend on any action taken by the 32-bit DLL's initialization code until at least one 16- to 32-bit thunk has been called.
- Missing 32-bit DLLs or failed 32-bit loads will not be detected until the first call is made to a 16- to 32-bit thunk. If the 32-bit DLL cannot load or fails to initialize, the 16- to 32-bit thunk call will return a value of zero. This error code may be changed on a thunk by thunk basis by including a **faulterrorcode=***dword*; line between the curly braces of a function call. For example, the following function instructs the thunk subsytem to return a - 1 from the thunk call if it is the first thunk call and the 32-bit DLL cannot finish loading.

```
int Sample(void) {
    faulterrorcode = -1;
}
```

Although late loading is a valuable optimization for a 16-bit DLL that can execute autonomously from its 32-bit partner, it does complicate error recovery. Late loading can be disabled by including the following line in the thunk script.

```
preload32 = true;
```

If you use this option, the 16-bit subsystem will be released during the loading of your 16-bit DLL, possibly causing other 16-bit code to be reentered. For this reason, the **preload32** statement is not available if your thunk script requires the **win31compat** statement. For more information about the **win31compat** statement, see Compatibility with Existing 16-Bit DLLs.

Although the thunk compiler supports a "**preload16**" keyword for future expansion, late loading of 16-bit DLLs is neither supported nor planned.

Troubleshooting

This section outlines two common problems when thunking is implemented and describes their likely causes.

- The 32-bit DLL will not load.

  The Windows 95 loader requires that the 16-bit DLL be marked as subsystem version 4.0 or greater for the **DllEntryPoint** function to run. In addition, **DllEntryPoint** must be exported using the name "DllEntryPoint" and must be marked RESIDENTNAME.

  To check the version number, run the Microsoft EXE File Header Utility (EXEHDR), using the following command line.

  ```
  exehdr -v <your-16-bit-DLL>
  ```

  The output should contain the following line.

  ```
  Operating system: Microsoft Windows - version 4.0
  ```

  If the subsystem version number is less than 4.0, you should use RC.EXE from the Win32 SDK to set the version of your 16-bit DLL correctly.

- Loading the 16-bit DLL does not load the 32-bit DLL.

  This is by design. A 32-bit DLL does not load into a process context until that process calls its first 16- to 32-bit thunk. For more information, see Late Loading.

  If this feature is incompatible with your DLL design, you may disable it by including the following line in your thunk script.

  ```
  preload32 = true;
  ```

Tool Help Library

The tool help library of functions make it easier for developers to obtain information about currently executing Microsoft® Win32®-based applications. These functions are designed to streamline the creation of Win32-hosted tools, specifically Windows-based debugging applications.

About Tool Help Functions

The following topics are discussed in this article:

Snapshots of the System

Snapshots are at the core of the tool help functions. A snapshot is a read-only copy of the current state of one or more of the following lists that reside in system memory: processes, threads, modules, and heaps.

Processes that use tool help functions access these lists from snapshots instead of directly from the operating system. The lists in system memory change when processes are started and ended, threads are created and destroyed, executable modules are loaded and unloaded from system memory, and heaps are created and destroyed. The use of information from a snapshot prevents inconsistencies. Otherwise, changes to a list could possibly cause a thread to incorrectly traverse the list or cause an access violation (a GP fault). For example, if an application traverses the thread list while other threads are created or terminated, information that the application is using to traverse the thread list might become outdated and could cause an error for the application traversing the list.

You can take a snapshot of the system memory by using the **CreateToolhelp32Snapshot** function. You can control the content of a snapshot by specifying one or more of the following values when calling this function: TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS, and TH32CS_SNAPTHREAD.

The TH32CS_SNAPHEAPLIST and TH32CS_SNAPMODULE values are process specific. When these values are specified, the heap and module lists of the specified process are included in the snapshot. If you specify zero as the process identifier, the current process is used. The TH32CS_SNAPTHREAD value always creates a system-wide snapshot even if a process identifier is passed to **CreateToolhelp32Snapshot**.

You can enumerate the heap or module state for all processes by specifying the TH32CS_SNAPALL value and the current process. Then, for each process in the snapshot that is not the current process, you can call **CreateToolhelp32Snapshot** again, specifying the process identifier and the TH32CS_SNAPHEAPLIST or TH32CS_SNAPMODULE value.

You can retrieve an extended error status code for **CreateToolhelp32Snapshot** by using the **GetLastError** function.

When your process finishes using a snapshot, you should destroy it by using the **CloseHandle** function. Not destroying a snapshot causes the process to leak memory until the process exits, at which time the system reclaims the memory.

Note The snapshot handle acts like a file handle and is subject to the same rules regarding which processes and threads it is valid in.

Process Walking

A snapshot that includes the process list contains information about each currently executing process. You can retrieve information for the first process in the list by using the **Process32First** function. After retrieving the first process in the list, you can traverse the process list for subsequent entries by using the **Process32Next** function. **Process32First** and **Process32Next** fill a **PROCESSENTRY32** structure with information about a process in the snapshot.

You can retrieve an extended error status code for **Process32First** and **Process32Next** by using the **GetLastError** function.

You can read the memory in a specific process into a buffer by using the **ToolhelpReadProcessMemory** function (or the **VirtualQueryEx** function).

Note The contents of the **th32ProcessID** and **th32ParentProcessID** members of **PROCESSENTRY32** are process identifiers and can be used with Win32 Application Programming Interface (API) elements.

Thread Walking

A snapshot that includes the thread list contains information about each thread of each currently executing process. You can retrieve information for the first thread in the list by using the **Thread32First** function. After retrieving the first thread in the list, you can retrieve information for subsequent threads by using the **Thread32Next** function. **Thread32First** and **Thread32Next** fill a **THREADENTRY32** structure with information about individual threads in the snapshot.

You can enumerate the threads of a specific process by taking a snapshot that includes the threads and then by traversing the thread list, keeping information about the threads that have the same process identifier as the specified process.

You can retrieve an extended error status code for **Thread32First** and **Thread32Next** by using the **GetLastError** function.

Module Walking

A snapshot that includes the module list for a specified process contains information about each module, executable file, or dynamic-link library (DLL), used by the specified process. You can retrieve information for the first module in the list by using the **Module32First** function. After retrieving the first module in the list, you can retrieve information for subsequent modules in the list by using the **Module32Next** function. **Module32First** and **Module32Next** fill a **MODULEENTRY32** structure with information about the module.

You can retrieve an extended error status code for **Module32First** and **Module32Next** by using the **GetLastError** function.

Note The module identifier, which is specified in the **th32ModuleID** member of **MODULEENTRY32**, has meaning only to the tool help functions. It is not a handle, nor is it usable by Win32 API elements.

Heap Lists and Heap Walking

A snapshot that includes the heap list for a specified process contains identification information for each heap associated with the specified process and detailed information about each heap. You can retrieve an identifier for the first heap of the heap list by using the **Heap32ListFirst** function. After retrieving the first heap in the list, you can traverse the heap list for subsequent heaps associated with the process by using the **Heap32ListNext** function. **Heap32ListFirst** and **Heap32ListNext** fill a **HEAPLIST32** structure with the process identifier, the heap identifier, and flags describing the heap.

You can retrieve information about the first block of a heap by using the **Heap32First** function. After retrieving the first block of a heap, you can retrieve information about subsequent blocks of the same heap by using the **Heap32Next** function. **Heap32First** and **Heap32Next** fill a **HEAPENTRY32** structure with information for the appropriate block of a heap.

You can retrieve an extended error status code for **Heap32ListFirst**, **Heap32ListNext**, **Heap32First**, and **Heap32Next** by using the **GetLastError** function.

Note The heap identifier, which is specified in the **th32HeapID** member of the **HEAPENTRY32** structure, has meaning only to the tool help functions. It is not a handle, nor is it usable by Win32 API elements.

Using the Tool Help Functions

This section contains examples demonstrating how to perform the following tasks:

- Access tool help functions.
- Take a snapshot and view the processes in the system address space.
- Traverse the threads listed in a snapshot.
- Traverse the modules listed for a specific process.

The following examples have been taken from the PVIEW95 application included in the Win32 SDK.

Note Each of the **HEAPENTRY32**, **HEAPLIST32**, **MODULEENTRY32**, **PROCESSENTRY32**, and **THREADENTRY32** structures that are used with the tool help functions has a **dwSize** member that must be initialized to the size of the structure before the structure is included in a call to a tool help function. The value of **dwSize** is used to indicate the version of the tool help functions used. If your application does not initialize **dwSize**, the tool help function will fail.

Accessing the Tool Help Functions

The tool help functions reside in the operating system kernel. The following example provides a platform-independent approach to accessing the tool help functions.

```
#include <tlhelp32.h> // needed for tool help declarations

// Type definitions for pointers to call tool help functions.
typedef BOOL (WINAPI *MODULEWALK)(HANDLE hSnapshot,
    LPMODULEENTRY32 lpme);
typedef BOOL (WINAPI *THREADWALK)(HANDLE hSnapshot,
    LPTHREADENTRY32 lpte);
typedef BOOL (WINAPI *PROCESSWALK)(HANDLE hSnapshot,
    LPPROCESSENTRY32 lppe);
typedef HANDLE (WINAPI *CREATESNAPSHOT)(DWORD dwFlags,
    DWORD th32ProcessID);

// File scope globals. These pointers are declared because of the need
// to dynamically link to the functions.  They are exported only by
// the Windows 95 kernel. Explicitly linking to them will make this
// application unloadable in Microsoft(R) Windows NT(TM) and will
// produce an ugly system dialog box.
static CREATESNAPSHOT pCreateToolhelp32Snapshot = NULL;
static MODULEWALK  pModule32First  = NULL;
static MODULEWALK  pModule32Next   = NULL;
static PROCESSWALK pProcess32First = NULL;
static PROCESSWALK pProcess32Next  = NULL;
static THREADWALK  pThread32First  = NULL;
static THREADWALK  pThread32Next   = NULL;

// Function that initializes tool help functions.
BOOL InitToolhelp32 (void)
{
    BOOL   bRet  = FALSE;
    HANDLE hKernel = NULL;

    // Obtain the module handle of the kernel to retrieve addresses of
    // the tool helper functions.
    hKernel = GetModuleHandle("KERNEL32.DLL");

    if (hKernel){
        pCreateToolhelp32Snapshot =
            (CREATESNAPSHOT)GetProcAddress(hKernel,
            "CreateToolhelp32Snapshot");

        pModule32First  = (MODULEWALK)GetProcAddress(hKernel,
            "Module32First");
        pModule32Next   = (MODULEWALK)GetProcAddress(hKernel,
            "Module32Next");

        pProcess32First = (PROCESSWALK)GetProcAddress(hKernel,
            "Process32First");
        pProcess32Next  = (PROCESSWALK)GetProcAddress(hKernel,
            "Process32Next");
```

```
        pThread32First  = (THREADWALK)GetProcAddress(hKernel,
            "Thread32First");
        pThread32Next   = (THREADWALK)GetProcAddress(hKernel,
            "Thread32Next");

        // All addresses must be non-NULL to be successful.
        // If one of these addresses is NULL, one of
        // the needed lists cannot be walked.
        bRet =  pModule32First && pModule32Next  && pProcess32First &&
                pProcess32Next && pThread32First && pThread32Next &&
                pCreateToolhelp32Snapshot;
    }
    else
        bRet = FALSE; // could not even get the module handle of kernel

    return bRet;
}
```

Taking a Snapshot and Viewing Processes

The following function takes a snapshot of the currently executing processes in the system and walks through the list recorded in the snapshot.

```
BOOL GetProcessList (VOID)
{
    HANDLE         hSnapshot = NULL;
    BOOL           bRet      = FALSE;
    PROCESSENTRY32 pe32      = {0};

    //  Take a snapshot of all processes currently in the system.
    hSnapshot = pCreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hProcessSnap == (HANDLE)-1)
        return (FALSE);

    //  Fill in the size of the structure before using it.
    pe32.dwSize = sizeof(PROCESSENTRY32);

    //  Walk the snapshot of the processes, and for each process, get
    //  information to display.
    if (pProcess32First(hProcessSnap, &pe32)) {
        BOOL          bGotModule = FALSE;
        MODULEENTRY32 me32       = {0};
        PINFO         pi         = {0};

        do {
            bGotModule = GetProcessModule(pe32.th32ProcessID,
                pe32.th32ModuleID, &me32, sizeof(MODULEENTRY32));
            if (bGotModule) {
                HANDLE hProcess;

                // Get the actual priority class.
                hProcess = OpenProcess (PROCESS_ALL_ACCESS,
                    FALSE, pe32.th32ProcessID);
                pi.dwPriorityClass = GetPriorityClass (hProcess);
                CloseHandle (hProcess);

                // Get the process's base priority value.
                pi.pcPriClassBase = pe32.pcPriClassBase;
                pi.pid            = pe32.th32ProcessID;
                pi.cntThreads     = pe32.cntThreads;
                lstrcpy(pi.szModName, me32.szModule);
                lstrcpy(pi.szFullPath, me32.szExePath);

                AddProcessItem(hListView, pi);
            }
        }
        while (pProcess32Next(hProcessSnap, &pe32));
        bRet = TRUE;
    }
    else
        bRet = FALSE;    // could not walk the list of processes
```

```
    // Do not forget to clean up the snapshot object.
    CloseHandle (hProcessSnap);
    return (bRet);
}
```

Traversing the Thread List

The following function takes a snapshot of the threads currently executing in the system and walks through the list recorded in the snapshot.

```
// Returns TRUE if the threads were successfully enumerated
//   and listed or FALSE if the threads could not be enumerated
//   or listed.
// hListView  - handle of the listview that lists thread information
// dwOwnerPID - identifier of the process whose threads are to
//   be listed
BOOL RefreshThreadList (HWND hListView, DWORD dwOwnerPID)
{
    HANDLE        hThreadSnap = NULL;
    BOOL          bRet        = FALSE;
    THREADENTRY32 te32        = {0};

    // Take a snapshot of all threads currently in the system.
    hThreadSnap = pCreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if (hThreadSnap == (HANDLE)-1)
        return (FALSE);

    // Clear the current contents of the thread list view
    // (which are now old).
    ListView_DeleteAllItems(g_hwndThread);

    // Fill in the size of the structure before using it.
    te32.dwSize = sizeof(THREADENTRY32);

    // Walk the thread snapshot to find all the threads of the process.
    // If the thread belongs to the process, add its information
    // to the display list.
    if (pThread32First(hThreadSnap, &te32)) {
        do {
            if (te32.th32OwnerProcessID == dwOwnerPID) {
                TINFO ti;

                ti.tid        = te32.th32ThreadID;
                ti.pidOwner   = te32.th32OwnerProcessID;
                ti.tpDeltaPri = te32.tpDeltaPri;
                ti.tpBasePri  = te32.tpBasePri;

                AddThreadItem(hListView, ti);
            }
        }
        while (pThread32Next(hThreadSnap, &te32));
        bRet = TRUE;
    }
    else
        bRet = FALSE;          // could not walk the list of threads

    // Do not forget to clean up the snapshot object.
    CloseHandle (hThreadSnap);
```

```
    return (bRet);
}
```

Traversing the Module List

The following function takes a snapshot of the modules in the address space of a specified process and retrieves information for a specific module from the list recorded in the snapshot.

```
// Returns TRUE if there is information about the specified module or
//   FALSE if it could not enumerate the modules in the process or
//   the module is not found in the process.
// dwPID - identifier of the process that owns the module to
//   retrieve information about.
// dwModuleID - tool help identifier of the module within the
//   process
// lpMe32 - structure to return data about the module
// cbMe32 - size of the buffer pointed to by lpMe32 (to ensure
//   the buffer is not over filled)
BOOL GetProcessModule (DWORD dwPID, DWORD dwModuleID,
        LPMODULEENTRY32 lpMe32, DWORD cbMe32)
{
    BOOL          bRet        = FALSE;
    BOOL          bFound      = FALSE;
    HANDLE        hModuleSnap = NULL;
    MODULEENTRY32 me32        = {0};

    // Take a snapshot of all modules in the specified process.
    hModuleSnap = pCreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwPID);
    if (hModuleSnap == (HANDLE)-1)
        return (FALSE);

    // Fill the size of the structure before using it.
    me32.dwSize = sizeof(MODULEENTRY32);

    // Walk the module list of the process, and find the module of
    // interest. Then copy the information to the buffer pointed
    // to by lpMe32 so that it can be returned to the caller.
    if (pModule32First(hModuleSnap, &me32)) {
        do {
            if (me32.th32ModuleID == dwModuleID) {
                CopyMemory (lpMe32, &me32, cbMe32);
                bFound = TRUE;
            }
        }
        while (!bFound && pModule32Next(hModuleSnap, &me32));

        bRet = bFound;   // if this sets bRet to FALSE, dwModuleID
                         // no longer exists in specified process
    }
    else
        bRet = FALSE;           // could not walk module list

    // Do not forget to clean up the snapshot object.
    CloseHandle (hModuleSnap);

    return (bRet);
}
```

Tool Help Reference

The following functions and structures are associated with the tool help services.

Tool Help Functions

The following functions are used with tool help services:

CreateToolhelp32Snapshot
Heap32First
Heap32ListFirst
Heap32ListNext
Heap32Next
Module32First
Module32Next
Process32First
Process32Next
Thread32First
Thread32Next
Toolhelp32ReadProcessMemory

Tool Help Structures

The following structures are used with tool help services:

HEAPENTRY32
HEAPLIST32
MODULEENTRY32
PROCESSENTRY32
THREADENTRY32

Virtual Machine Services

This article describes the virtual machine services and shows how to use them in MS-DOS - based applications.

About Virtual Machine Services

Virtual machine services allow Microsoft® MS-DOS® - based applications to take advantage of features provided by Microsoft® Windows® 95 when the applications run in a window. MS-DOS - based applications can retrieve and, optionally, set the title of the window in which they run.

Window Title

The window title for an MS-DOS - based application, which is displayed when the application runs in a window, identifies the application and its operating state. The operating system sets the title when an application first starts, but the application can change portions of the title to better communicate its state to the user.

The window title consists of three strings: a virtual machine state, a virtual machine title, and an application title. The system creates the window title by concatenating the three strings, separating the strings with a system-defined separator, typically a hyphen.

The virtual machine state, which can be set by the operating system only, identifies whether the virtual machine is inactive or whether the user is carrying out tasks, such as cut and paste operations. The state is frequently an empty string. The virtual machine title and the application title, which can be set by an application, identify the application and the current document or activity of the application, respectively.

By default, the operating system sets the virtual machine title to the title stored in the corresponding .PIF file or to a title specified by the shell. You can determine the current virtual machine title by using the Get Virtual Machine Title function. You can change the title by using the Set Virtual Machine Title function.

The system typically sets the application title to the name of the application, so the virtual machine title and application title are frequently the same when the application first starts. Many applications change this title to the name of the current document or to an empty string if there is no current document. You can determine the current application title by using the Get Application Title function. You can change the title by using the Set Application Title function.

If you set the virtual machine title or application title, the length of the individual titles must not exceed 30 and 80 characters, respectively.

You can call the window title functions at any time. Furthermore, you can call these functions regardless of whether your application is running with the Windows 95 or MS-DOS operating system. However, not all operating systems support these calls. The functions are not supported if a call to a function leaves the AX register unchanged.

Close-Aware Applications

A close-aware application is any MS-DOS - based application that periodically checks the state of an internal close flag and terminates if the flag is set. Windows 95 sets this flag when the user chooses the Close command from the system menu of the window in which the MS-DOS - based application runs. Close-aware applications enable the Close command, which gives the user an alternate way to exit the application and close the window.

An application enables or disables the Close command by using the Enable or Disable Close Command function. The function takes a flag indicating whether to enable or disable the command. Once the command is enabled, the application must periodically check the close flag by using the Query Close function. The function returns zero in the AX register if the user has chosen the command.

If the Query Close function returns zero in AX, the application should call the Acknowledge Close function to acknowledge the close state of the internal close flag. After the application acknowledges the close state, subsequent calls to the Query Close function will return 1 in AX, indicating that the user has chosen the Close command and the close state has been acknowledged. After acknowledging the close state, an application should take all necessary steps to shut down and eventually exit, or it should cancel the close operation by calling the Cancel Close function.

An application should acknowledge the close state if it needs to perform additional keyboard input before exiting. When the close state has been signaled but has not yet been acknowledged by the application, all keyboard reads will return NULL and buffered line input will return an empty string.

After an application acknowledges the close state, the state reverts to unacknowledged if the application either exits or cancels the close operation. If the application acknowledges the close state and then exits, the parent process will be in an unacknowledged close state. The application must then acknowledge the close state to perform additional keyboard input before exiting or canceling the close operation.

For example, if a text editor receives a positive response from the Query Close function and has some buffers that have not been saved, it should call Acknowledge Close and ask the user if the buffers should be saved with these possible responses: "Yes," "No," or "Cancel."

If the user responds "Yes" or "No," the text editor should save (or not save if the response was "No") the buffers and then exit. The close state remains active, and the parent process (probably the command interpreter) will also receive a positive response from Query Close and will also terminate.

If the response is "Cancel," the application should call the Cancel Close function and not exit. Canceling the close operation informs the system that any attempted shutdown should be abandoned.

This sequence of operations is analogous to the way that Windows-based applications handle the WM_QUERYENDSESSION message.

Depending on the tasking option chosen for the application, there may be some time between when the user chooses the Close command and the application checks the internal close flag. During this time, the system changes the window title of the application, appending the word "Closing" to it, and gives the user the opportunity to cancel the command by changing the command name to Cancel Close. If the user chooses the Cancel Close command, the close flag is reset, preventing the application from closing. If a close-aware application fails to check the close flag within a system-defined amount of time, the system automatically abandons the operation and resets the close flag.

The system tracks the close-awareness and close state for each process. For the virtual machine to close, all applications in the virtual machine must close. When the user chooses the Close command, the operating system detects applications that are not close-aware and displays a dialog box with a warning message, but gives the user the option of forcing the application to exit anyway. If you make an application close-aware, it can shut down cleanly.

Virtual Machine Services Reference

The following window title and close-aware functions are used with virtual machine services.

Window Title Functions

The following window title functions are used with virtual machine services:

Get Application Title
Get Virtual Machine Title
Set Application Title
Set Virtual Machine Title

Close-Aware Application Functions

The following close-aware application functions are used with virtual machine services:

Acknowledge Close
Cancel Close
Enable or Disable Close Command
Query Close

Windows 95 Service Control Manager

Microsoft® Windows NT® supports a Win32-based application type known as a service. A service may be started in two ways: automatically, when the system starts up; or upon demand, by Win32-based applications that use the functions provided by the Service Control Manager. For more information, see Services.

The Windows NT Service Control Manager and its associated functions are not supported by Windows 95®. Instead, Windows 95 provides a scaled-down Service Control Manager.

About the Windows 95 Service Control Manager

The Windows 95 Service Control Manager uses two registry keys to allow applications to start at system startup. It uses one function to allow an application to continue to run after the user logs off.

- The **RunServices** and **RunServicesOnce** Keys
- The **RegisterServiceProcess** Function

The RunServices and RunServicesOnce Keys

During system startup, the Service Control Manager starts the applications listed under the following registry keys: **RunServices** and **RunServicesOnce**. These keys are located under the key

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion

You can use any unique value name for each application. For the value data, use a string that represents the command line the Service Control Manager will use to start the application. Following is an example of a **RunServices** list that contains entries for three applications:

MyApp1=myapp1.exe
MyApp2=myapp2.exe *param1 param2*
MyApp3=c:\mydir\myapp3.exe

The system deletes values under the key **RunServicesOnce** after the application starts.

These applications start before the user logs on. The user is not yet validated, and the applications cannot assume that networking permissions are enabled. Until the user logs on, the application does not have access to network resources that the user has access to.

The RegisterServiceProcess Function

Applications started by the system using the **RunServices** and **RunServicesOnce** keys will close when the user selects **Close all programs and log on as a different user** from the **Shutdown** dialog box. By calling the **RegisterServiceProcess** function, a Win32-based application can prevent itself or any other Win32-based application from being closed when the user logs off. Win32-based applications registered in this manner close only when the system is shut down.

The application should provide for different users logging on at different times during its execution. The application can distinguish between a user logging off and the system shutting down by examining the *lParam* parameter of the WM_QUERYENDSESSION and WM_ENDSESSION messages. If the user shuts down the system, *lParam* is NULL. If the user logs off, *lParam* is set to EWX_REALLYLOGOFF.

To call **RegisterServiceProcess**, retrieve a function pointer using **GetProcAddress** on KERNEL32.DLL. Use the function pointer to call **RegisterServiceProcess**.

Windows 95 Service Control Manager Reference

The following functions are supported by the Windows 95 Service Control Manager.

Windows 95 Service Control Manager Functions

The following function is supported by the Windows 95 Service Control Manager.

RegisterServiceProcess

Acknowledge Close Overview

Group

Acknowledges the close state of the internal close flag.

```
mov ah, 16h    ; Windows multiplex function
mov al, 8Fh    ; VM Close
mov dh, 2      ; Acknowledge Close
mov dl, 0      ; always 0
int 2Fh
or ax, ax
jz success
```

Return Value

Returns zero in the AX register if successful.

Remarks

Acknowledging the close state is necessary if an application needs to perform additional keyboard input before exiting. If the close state has been signaled but not yet acknowledged by an application, all keyboard read operations will return NULL and buffered line input will return an empty string.

Cancel Close Overview

Group

Cancels the close operation.

```
mov ah, 16h    ; Windows multiplex function
mov al, 8Fh    ; VM Close
mov dh, 3      ; Cancel Close
mov dl, 0      ; always 0
int 2Fh
or ax, ax
jz success
```

Return Value

Returns zero in the AX register if successful.

Remarks

After acknowledging the close state of the internal close flag, an application should either exit or cancel the close operation by calling Cancel Close.

CreateToolhelp32Snapshot Overview

Group

Takes a snapshot of the processes and the heaps, modules, and threads used by the processes.

```
HANDLE WINAPI CreateToolhelp32Snapshot(DWORD dwFlags,
    DWORD th32ProcessID);
```

Parameters

dwFlags

Flags specifying portions of the system to include in the snapshot. These values are defined:

TH32CS_INHERIT	Indicates that the snapshot handle is to be inheritable.
TH32CS_SNAPALL	Equivalent to specifying the TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS, and TH32CS_SNAPTHREAD values.
TH32CS_SNAPHEAPLIST	Includes the heap list of the specified process in the snapshot.
TH32CS_SNAPMODULE	Includes the module list of the specified process in the snapshot.
TH32CS_SNAPPROCESS	Includes the Win32 process list in the snapshot.
TH32CS_SNAPTHREAD	Includes the Win32 thread list in the snapshot.

th32ProcessID

Process identifier. This parameter can be zero to indicate the current process. This parameter is used when the TH32CS_SNAPHEAPLIST or TH32CS_SNAPMODULE value is specified. Otherwise, it is ignored.

Return Value

Returns an open handle to the specified snapshot if successful or - 1 otherwise.

Remarks

The snapshot taken by this function is examined by the other tool help functions to provide their results. Access to the snapshot is read only. The snapshot handle acts like an object handle and is subject to the same rules regarding which processes and threads it is valid in.

To retrieve an extended error status code generated by this function, use the **GetLastError** function.

To destroy the snapshot, use the **CloseHandle** function.

Enable or Disable Close Command Overview

Group

Enables or disables the Close command in the system menu.

```
mov ah, 16h    ; Windows multiplex function
mov al, 8Fh    ; VM Close
mov dh, 0      ; Enable or Disable Close Command
mov dl, Flags  ; see below
int 2Fh
or ax, ax
jz success
```

Parameters

Flags

Close flags. This parameter can be one of these values:

00h Disables the Close command.

01h Enables the Close command.

Return Value

Returns zero in the AX register if successful.

Get Application Title Overview

Group

Copies the application title to the specified buffer.

```
mov  ah, 16h           ; Windows multiplex function
mov  al, 8Eh           ; VM Title
mov  di, seg AppTitle  ; see below
mov  es, di
mov  di, offset AppTitle
mov  cx, Size          ; see below
mov  dx, 2             ; Get Application Title
int  2Fh
cmp  ax, 1
je   success
```

Parameters

AppTitle

Address of a buffer that receives the application title. This parameter must not be zero.

Size

Size, in bytes, of the buffer pointed to by *AppTitle*.

Return Value

Returns 1 in the AX register if successful or zero otherwise.

Remarks

Get Application Title copies as much of the title as possible, but never more than the specified number of bytes. The function always appends a terminating null character to the title in the buffer.

Get Virtual Machine Title Overview

Group

Copies the virtual machine title to the specified buffer.

```
mov  ah, 16h           ; Windows multiplex function
mov  al, 8Eh           ; VM Title
mov  di, seg VMTitle   ; see below
mov  es, di
mov  di, offset VMTitle
mov  cx, Size          ; see below
mov  dx, 3             ; Get Virtual Machine Title
int  2Fh
cmp  ax, 1
je   success
```

Parameters

VMTitle

Address of a buffer that receives the virtual machine title. This parameter must not be zero.

Size

Size, in bytes, of the buffer pointed to by *VMTitle*.

Return Value

Returns 1 in the AX register if successful or zero otherwise.

Remarks

Get Virtual Machine Title copies as much of the title as possible, but never more than the specified number of bytes. The function always appends a terminating null character to the title in the buffer.

Heap32First Group

Group

Retrieves information about the first block of a heap that has been allocated by a process.

```
BOOL WINAPI Heap32First(LPHEAPENTRY32 lphe, DWORD th32ProcessID,
    DWORD th32HeapID);
```

Parameters

lphe
Address of a buffer containing a **HEAPENTRY32** structure.

th32ProcessID
Identifier of the process context that owns the heap.

th32HeapID
Identifier of the heap to enumerate.

Return Value

Returns TRUE if information for the first heap block has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if the heap is invalid or empty.

Remarks

The calling application must set the **dwSize** member of **HEAPENTRY32** to the size, in bytes, of the structure. **Heap32First** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To access subsequent blocks of the same heap, use the **Heap32Next** function.

Heap32ListFirst Group

Group

Retrieves information about the first heap that has been allocated by a specified process.

```
BOOL WINAPI Heap32ListFirst(HANDLE hSnapshot, LPHEAPLIST32 lphl);
```

Parameters

hSnapshot

Handle of the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lphl

Address of a buffer containing a **HEAPLIST32** structure.

Return Value

Returns TRUE if the first entry of the heap list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function when no heap list exists or the snapshot does not contain heap list information.

Remarks

The calling application must set the **dwSize** member of **HEAPLIST32** to the size, in bytes, of the structure. **Heap32ListFirst** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To retrieve information about other heaps in the heap list, use the **Heap32ListNext** function.

Heap32ListNext Group

Group

Retrieves information about the next heap that has been allocated by a process.

```
BOOL WINAPI Heap32ListNext(HANDLE hSnapshot, LPHEAPLIST32 lphl);
```

Parameters

hSnapshot

Handle of the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lphl

Address of a buffer containing a **HEAPLIST32** structure.

Return Value

Returns TRUE if the next entry of the heap list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function when no more entries in the heap list exist.

Remarks

To retrieve information about the first heap in a heap list, use the **Heap32ListFirst** function.

Heap32Next Group

Group

Retrieves information about the next block of a heap that has been allocated by a process.

```
BOOL WINAPI Heap32Next(LPHEAPENTRY32 lphe);
```

Parameters

lphe

Address of a buffer containing a **HEAPENTRY32** structure.

Return Value

Returns TRUE if information about the next block in the heap has been copied to the buffer or FALSE otherwise. The **GetLastError** function returns ERROR_NO_MORE_FILES when no more objects in the heap exist and ERROR_INVALID_DATA if the heap appears to be corrupt or is modified during the walk in such a way that **Heap32Next** cannot continue.

Remarks

To retrieve information for the first block of a heap, use the **Heap32First** function.

Interrupt 21h Function 4302h Group

Group

Obtains the compressed size, in bytes, of a given file or directory.

```
mov ax, 4302h            ; Get Compressed File Size
mov dx, seg PathName     ; see below
mov ds, dx
mov dx, offset PathName
int 21h

jc error
```

Parameters

PathName

Address of a null-terminated string that specifies the file or directory to retrieve the file size for.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

The function obtains the actual number of bytes of disk storage used to store the file. If the file is located on a volume that supports compression and the file is compressed, the value obtained is the compressed size of the specified file. If the file is not located on a volume that supports compression or if the file is not compressed, the value obtained is the file size, in bytes, rounded up to the nearest cluster boundary.

Interrupt 21h Function 440Dh Minor Code 48h

Group

Group

Locks or unlocks the volume in the given drive (preventing or permitting its removal) or returns the locked status of the given drive.

```
mov ax, 440Dh           ; generic IOCTL
mov bx, DriveNum        ; see below
mov ch, 8               ; device category
mov cl, 48h             ; Lock or Unlock Removable Media
mov dx, seg ParamBlock  ; see below
mov ds, dx
mov dx, offset ParamBlock
int 21h

jc  error
```

Parameters

DriveNum

Drive to lock or unlock. This parameter can be 0 for default drive, 1 for A, 2 for B, and so on.

ParamBlock

Address of a **PARAMBLOCK** structure that specifies the operation to carry out and receives a count of the number of locks on the drive.

Return Value

Clears the carry flag and copies the number of pending locks on the given drive to the **NumLocks** member of the **PARAMBLOCK** structure if successful. Otherwise, the function sets the carry flag and sets the AX register to one of the following error values:

01h	The function is not supported.
B0h	The volume is not locked in the drive.
B2h	The volume is not removable.
B4h	The lock count has been exceeded.

Interrupt 21h Function 440Dh Minor Code 49h

Group

Group

Ejects the specified media.

```
mov ax, 440Dh     ; generic IOCTL
mov bx, DriveNum  ; see below
mov ch, 8         ; device category
mov cl, 49h       ; Eject Removable Media
int 21h

jc  error
```

Parameters

DriveNum

Drive to eject. This parameter can be 0 for default drive, 1 for A, 2 for B, and so on.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to one of the following error values:

01h	The function is not supported.
B1h	The volume is locked in the drive.
B2h	The volume is not removable.
B5h	The valid eject request has failed.

Remarks

If a given physical drive has more than one logical volume, all volumes must be unlocked by using Lock/Unlock Removable Media (Interrupt 21h Function 440Dh Minor Code 48h) before the drive will eject.

Interrupt 21h Function 440Dh Minor Code 4Ah

Group

Group

Locks the logical volume.

```
mov ax, 440Dh        ; generic IOCTL
mov bh, LockLevel    ; see below
mov bl, DriveNum     ; see below
mov ch, 08h          ; device category (must be 08h)
mov cl, 4Ah          ; Lock Logical Volume
mov dx, Permissions  ; see below
int 21h

jc error
```

Parameters

LockLevel

Level of the lock. This parameter must be either 0, 1, 2, or 3.

DriveNum

Drive to lock. This parameter can be 0 for the default drive, 1 for A, 2 for B, and so on.

Permissions

Operations that the system permits while the volume is locked. This parameter is specified only when a level 1 lock is obtained or when a level 0 lock is obtained for the second time for formatting the volume. For other lock levels, this parameter is zero. When a level 1 lock is obtained, bits 0 and 1 of this parameter specify whether the system permits write operations, new file mappings, or both by other processes during a level 1 lock as well as during level 2 and 3 locks. If this parameter specifies that write operations, new file mappings, or both are failed, these operations are failed during level 1, 2, and 3 locks. This parameter has the following form:

Bit	Meaning
0	0 = Write operations are failed (specified when a level 1 lock is obtained).
0	1 = Write operations are allowed (specified when a level 1 lock is obtained).
1	0 = New file mapping are allowed (specified when a level 1 lock is obtained).
1	1 = New file mapping are failed (specified when a level 1 lock is obtained).
2	1 = The volume is locked for formatting (specified when a level 0 lock is obtained for the second time).

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

The volume must be locked before the application performs direct disk write operations by using Interrupt 26h or the IOCTL control functions. Lock Physical Volume (Interrupt 21h Function 440Dh Minor Code 4Bh) is used instead of this function before a call to an Interrupt 13h function. Unlock Logical Volume (Interrupt 21h Function 440Dh Minor Code 6Ah) should be used to release the lock.

Interrupt 21h Function 440Dh Minor Code 4Bh

Group

Group

Locks the physical volume.

```
mov ax, 440Dh        ; generic IOCTL
mov bh, LockLevel    ; see below
mov bl, DriveNum     ; see below
mov ch, 08h          ; device category (must be 08h)
mov cl, 4Bh          ; Lock Physical Volume
mov dx, Permissions  ; see below
int 21h

jc error
```

Parameters

LockLevel

Level of the lock. This parameter must be either 0, 1, 2, or 3.

DriveNum

Drive to lock. This parameter must be one of these values (same device unit numbers as for Interrupt 13h):

00 - 7Fh	Floppy disk drive (00 for the first floppy drive, 01 for the second, and so on).
80 - FFh	Hard disk drive (80 for the first hard disk drive, 81 for the second, and so on).

Permissions

Operations that the system permits while the volume is locked. This parameter is specified only when a level 1 lock is obtained or when a level 0 lock is obtained for the second time for formatting the volume. For other lock levels, this parameter is zero. When a level 1 lock is obtained, bits 0 and 1 of this parameter specify whether the system permits write operations, new file mappings, or both by other processes during a level 1 lock as well as during level 2 and 3 locks. If this parameter specifies that write operations, new file mappings, or both are failed, these operations are failed during level 1, 2, and 3 locks. This parameter has the following form:

Bit	Meaning
0	0 = Write operations are failed (specified when a level 1 lock is obtained).
0	1 = Write operations are allowed (specified when a level 1 lock is obtained).
1	0 = New file mapping are allowed (specified when a level 1 lock is obtained).
1	1 = New file mapping are failed (specified when a level 1 lock is obtained).
2	1 = The volume is locked for formatting (specified when a level 0 lock is obtained for the second time).

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

The volume must be locked before the application performs direct disk write operations by using Interrupt 13h, Interrupt 26h, or the Interrupt 21h IOCTL functions. A single physical volume may be divided into more than one logical volume, which is also called a partition. The system automatically takes a logical volume lock on all logical volumes on the specified physical drive. If the application performs disk writes only to a logical drive, Lock Logical Volume (Interrupt 21h Function 440Dh Minor Code 4Ah) is used instead of this function. Unlock Physical Volume (Interrupt 21h Function 440Dh Minor Code 6Bh) should be called to release the lock.

Interrupt 21h Function 440Dh Minor Code 6Ah

Group

Group

Unlocks the logical volume or decrements the lock level.

```
mov ax, 440Dh      ; generic IOCTL
mov bl, DriveNum   ; see below
mov ch, 08h        ; device category (must be 08h)
mov cl, 6Ah        ; Unlock Logical Volume
int 21h

jc error
```

Parameters

DriveNum

Drive to unlock. This parameter can be 0 for the default drive, 1 for A, 2 for B, and so on.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

This function is used to release the lock obtained by using Lock Logical Volume (Interrupt 21h Function 440Dh Minor Code 4Ah). Only the lock owner can release the lock on a volume.

To release the lock on the volume, an application must call Unlock Logical Volume the same number of times that Lock Logical Volume was called.

Interrupt 21h Function 440Dh Minor Code 6Bh

Group

Group

Unlocks the physical volume or decrements the lock level.

```
mov ax, 440Dh      ; generic IOCTL
mov bl, DriveNum   ; see below
mov ch, 08h        ; device category (must be 08h)
mov cl, 6Bh        ; Unlock Physical Volume
int 21h

jc enter
```

Parameters

DriveNum

Drive to unlock. This parameter must be one of these values (same device unit numbers as for Interrupt 13h):

00 - 7Fh	Floppy disk drive (00 for the first floppy drive, 01 for the second, and so on).
80 - FFh	Hard disk drive (80 for the first hard disk drive, 81 for the second, and so on).

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

This function is used to release the lock obtained by using Lock Physical Volume (Interrupt 21h Function 440Dh Minor Code 4Bh). Only the lock owner can release the lock on a volume.

To release the lock on the volume, an application must call Unlock Physical Volume the same number of times that Lock Physical Volume was called.

Interrupt 21h Function 440Dh Minor Code 6Ch

Group

Group

Polls the state of the access flag on a volume to determine if a write operation (for example, deleting or renaming a file or writing to a file) or a new file mapping has occurred since the last poll.

```
mov ax, 440Dh        ; generic IOCTL
mov bl, DriveNum     ; see below
mov ch, 08h          ; device category (must be 08h)
mov cl, 6Ch          ; Get Lock Flag State
int 21h

jc error
mov [AccessFlag], ax  ; state of access flag
```

Parameters

DriveNum

Drive to poll. This parameter can be 0 for the default drive, 1 for A, 2 for B, and so on.

Return Value

Clears the carry flag and sets the AX register to one of the following values if successful:

0	No write operations or file mappings have occurred since the last poll.
1	A write operation has occurred since the last poll (clears the volume access flag).
2	A file mapping has occurred since the last poll, or a 32-bit Windows-based DLL or executable has been opened (clears the volume access flag).

Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

Only the current lock owner may poll the access flag. The system fails other processes with ERROR_ACCESS_DENIED error value. Write operations performed by the lock owner do not cause a change in the state of the access flag.

When a lock is obtained that allows write operations or new file mappings, the system sets a flag whenever one of these operations happens on the volume. If a write operation or new file mapping has occurred since the last poll, Get Lock Flag State returns 1 or 2 respectively in the AX register and clears the volume access flag. If the swap file has grown or shrunk since the last poll, Get Lock Flag State returns 1. Note that write operations to the swap file that do not cause a change in size do not cause a change in the state of the access flag. If a 32-bit Windows-based DLL or executable has been opened since the last poll, Get Lock Flag State returns 2.

Interrupt 21h Function 440Dh Minor Code 6Dh

Group

Group

Enumerates open files on the specified drive.

```
mov ax, 440Dh           ; generic IOCTL
mov bx, DriveNum        ; see below
mov ch, 08h             ; device category (must be 08h)
mov cl, 6Dh             ; Enumerate Open Files
mov dx, seg PathBuf     ; see below
mov ds, dx
mov dx, offset PathBuf
mov si, FileIndex       ; see below
mov di, EnumType        ; see below
int 21h

jc  error
mov [OpenMode], ax      ; mode file was opened in
mov [FileType], cx      ; normal file or memory-mapped file
```

Parameters

DriveNum

Drive on which to enumerate the files. This parameter can be 0 for the default drive, 1 for A, 2 for B, and so on.

PathBuf

Address of a buffer that receives the path of the open file. The length of the buffer varies depending on the volume. Get Volume Information (Interrupt 21h Function 71A0h) is used to determine the maximum allowed length of a path for the volume.

FileIndex

Index of the file to retrieve the path for.

EnumType

Kind of file to enumerate. This parameter can be 0 to enumerate all open files or 1 to enumerate only open unmovable files, including open memory-mapped files and other open unmovable files (32-bit Windows-based DLLs and executables).

Return Value

Clears the carry flag, copies the path of an open file to the given buffer, and sets the AX and CX registers to the following values if successful:

AX — Mode that the file was opened in, which is a combination of access mode, sharing mode, and open flags. It can be one value each from the access and sharing modes and any combination of open flags.

Access modes

OPEN_ACCESS_READONLY (0000h)

OPEN_ACCESS_WRITEONLY (0001h)

OPEN_ACCESS_READWRITE (0002h)

OPEN_ACCESS_RO_NOMODLASTACCESS (0004h)

Share modes

OPEN_SHARE_COMPATIBLE (0000h)

OPEN_SHARE_DENYREADWRITE (0010h)
OPEN_SHARE_DENYWRITE (0020h)
OPEN_SHARE_DENYREAD (0030h)
OPEN_SHARE_DENYNONE (0040h)

Open flags

OPEN_FLAGS_NOINHERIT (0080h)
OPEN_FLAGS_NO_BUFFERING (0100h)
OPEN_FLAGS_NO_COMPRESS (0200h)
OPEN_FLAGS_ALIAS_HINT (0400h)
OPEN_FLAGS_NOCRITERR (2000h)
OPEN_FLAGS_COMMIT (4000h)

CX File type. It can be one of the following values:

- 0 For normal files
- 1 For a memory-mapped files (memory-mapped files are unmovable)
- 2 For any other unmovable files (32-bit Windows-based DLLs and executables)
- 4 For the swap file

Note that if a memory-mapped file is returned (CX = 1), the value returned in the AX register is limited to the following values:

OPEN_ACCESS_READONLY (0000h)
OPEN_ACCESS_READWRITE (0002h)

Otherwise, the function sets the carry flag and sets the AX register to the following error value:

ERROR_ACCESS_DENIED	The value of *FileIndex* exceeds the number of open files on the drive.

Remarks

This function returns information about one file at a time. To enumerate all open files, the function must be called repeatedly with *FileIndex* set to a new value for each call. *FileIndex* should be set to zero initially and then incremented by one for each subsequent call. The function returns the ERROR_NO_MORE_FILES error value when all open files on the volume have been enumerated.

This function may return inconsistent results when used to enumerate files on an active volume – that is, on a volume where other processes may be opening and closing files. Applications should use Lock Logical Volume (Interrupt 21h Function 440Dh Minor Code 4Ah) to take a level 3 lock before enumerating open files.

Interrupt 21h Function 440Dh Minor Code 6Eh

Group

Group

Retrieves information about the swap file.

```
mov ax, 440Dh           ; generic IOCTL
mov ch, 08h             ; device category (must be 08h)
mov cl, 6Eh             ; Find Swap File
mov dx, seg PathBuf     ; see below
mov ds, dx
mov dx, offset PathBuf
int 21h

jc  error
mov [PagerType], ax            ; pager type
mov WORD PTR [FileSize], bx    ; swap file size in 4K pages
mov WORD PTR [FileSize+2], cx
```

Parameters

PathBuf

Address of the buffer that receives the path of the swap file. To determine the maximum allowed length of a path for the volume, call Get Volume Information (Interrupt 21h Function 71A0h).

Return Value

Clears the carry flag, copies the swap file path to the given buffer, and sets the following registers if successful:

AX	Pager type. It can be 1 for no pager, 2 for paging through MS-DOS, and 3 for paging through the protected-mode input and output (I/O) supervisor.
CX:BX	Current size of the swap file in 4K pages.

Otherwise, this function sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 440Dh Minor Code 6Fh

Group

Group

Retrieves information about the specified drive.

```
mov ax, 440Dh            ; generic IOCTL
mov bx, DriveNum         ; see below
mov ch, 8                ; device category
mov cl, 6Fh              ; Get Drive Map Info
mov dx, seg DriveMapInfo ; see below
mov ds, dx
mov dx, offset DriveMapInfo
int 21h

jc  error
```

Parameters

DriveNum

Drive to obtain information about. This parameter can be 0 for the default drive, 1 for A, 2 for B, and so on.

DriveMapInfo

Address of the **DRIVE_MAP_INFO** structure that receives information about the specified drive.

Interrupt 21h Function 440Dh Minor Code 70h

Group

Group

Retrieves the current lock level and permissions on the specified drive.

```
mov ax, 440Dh        ; generic IOCTL
mov bl, DriveNum     ; see below
mov ch, 08h          ; device category (must be 08h)
mov cl, 70h          ; Get Current Lock State
int 21h

jc error
```

Parameters

DriveNum

Drive to retrieve lock information about. This parameter can be 0 for the default drive, 1 for A, 2 for B, and so on.

Return Value

Clears the carry flag and sets the AX and CX registers to these values if successful:

AX Current lock level. It may be either 0, 1, 2 or 3. If the volume is not locked, AX contains - 1

CX Lock permissions. The bits have the following form:

Bit	Meaning
0	0 = Write operations are failed.
0	1 = Write operations are allowed, unless they are blocked by the lock level.
1	0 = New file mapping are allowed, unless they are blocked by the lock level.
1	1 = New file mapping are failed.
2	1 = The volume is locked for formatting.

Remarks

The lock level and the permissions determine the kind of access processes other than the lock owner have to the volume while it is locked. The following operations are allowed by processes other than lock owner at each lock level:

Level	Operations
0	Read operations, write operations, and new file mappings are failed.
1	Read operations are allowed. Write operations and new file mappings are either allowed or failed based on permissions.
2	Read operations are allowed. Write operations and new file mappings are either failed or blocked based on permissions.
3	Read operations are blocked. Write operations and new file mappings are either failed or blocked based on

permissions.

Interrupt 21h Function 440Dh Minor Code 71h

Group

Group

Retrieves the first cluster of the specified file or directory.

```
mov ax, 440Dh            ; generic IOCTL
mov bx, CharSet          ; see below
mov ch, 08h              ; device category
mov cl, 71h              ; Get First Cluster
mov dx, seg PathName     ; see below
mov ds, dx
mov dx, offset PathName
int 21h

jc error
```

Parameters

CharSet

Character set of *PathName*. This parameter must be one of these values:

BCS_WANSI (0)	Windows ANSI character set
BCS_OEM (1)	Current OEM character set
BCS_UNICODE (2)	Unicode character set

PathName

Address of a null-terminated string containing the path of the file or directory to retrieve the first cluster for.

Return Value

Clears the carry flag and sets DX:AX to the first cluster number if successful. Otherwise, the function sets the carry flag and returns either the ERROR_INVALID_FUNCTION or ERROR_ACCESS_DENIED value in AX.

Remarks

The first cluster of a file is the first cluster of the FAT cluster chain describing the data associated with the file. The first cluster of a directory is the first cluster of the FAT cluster chain associated with the directory. It is the cluster that contains the "." and ".." entries. The function finds any file or directory regardless of attribute (system, hidden, or read-only). It does not find volume labels.

If your application is unable to accommodate a 32-bit cluster number, you must check to see if the value returned in the DX register is greater than zero.

```
if(MAKELONG(regAX,regDX) > 0x0000FFF8)
    b32BitNum = TRUE;
else
    b32BitNum = FALSE;
```

It is the calling application's responsibility to check to see if the returned cluster number is valid.

```
if((MAKELONG(regAX,regDX) < 2L) || (MAKELONG(regAX,regDX) > maxClus))
    bInvalidNum = TRUE;
else
    bInvalidNum = FALSE;
```

In the preceding example, the *maxClus* variable is the maximum legal cluster number, as a **DWORD** type, computed from the drive parameters.

Interrupt 21h Function 5704h Group

Group

Retrieves the last access date for the given file.

```
mov ax, 5704h   ; Get Last Access Date and Time
mov bx, Handle  ; see below
int 21h

jc  error
mov [Date], dx  ; last access date
mov [Time], cx  ; currently not supported, always 0
```

Parameters

Handle
File handle.

Return Value

Clears the carry flag and sets the CX register to zero and the DX register to these values if successful:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (that is, add 1980 to get the actual year)

Otherwise, the function sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 5705h Group

Group

Sets the last access date for the given file.

```
mov ax, 5705h       ; Set Last Access Date and Time
mov bx, Handle      ; see below
mov cx, 0           ; time currently not supported, always 0
mov dx, AccessDate  ; see below
int 21h

jc  error
```

Parameters

Handle

File handle.

AccessDate

New access date. The date is a packed 16-bit value with this form:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (that is, add 1980 to get the actual year)

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 5706h Group

Group

Retrieves the creation date and time for the given file.

```
mov ax, 5706h           ; Get Creation Date and Time
mov bx, Handle          ; see below
int 21h

jc  error
mov [Time], cx          ; creation time
mov [Date], dx          ; creation date
mov [MilliSeconds], si  ; number of 10 ms intervals in 2 seconds
```

Parameters

Handle

File handle.

Return Value

Clears the carry flag and sets the CX, DX, and SI registers to these values if successful:

CX — Creation time. The time is a packed 16-bit value with the following form:

Bits	Contents
0-4	Second divided by 2
5-10	Minute (0-59)
11-15	Hour (0-23 on a 24-hour clock)

DX — Creation date. The date is a packed 16-bit value with the following form:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (that is, add 1980 to get the actual year)

SI — Number of 10 millisecond intervals in 2 seconds to add to the MS-DOS time. The number can be a value in the range of 0 to 199.

Otherwise, the function sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 5707h Group

Group

Sets the creation date and time for the given file.

```
mov ax, 5707h         ; Set Creation Date and Time
mov bx, Handle        ; see below
mov cx, Time          ; see below
mov dx, Date          ; see below
mov si, MilliSeconds  ; see below
int 21h

jc  error
```

Parameters

Handle

File handle.

Time

New creation time. The time is a packed 16-bit value with the following form:

Bits	Contents
0-4	Second divided by 2
5-10	Minute (0-59)
11-15	Hour (0-23 on a 24-hour clock)

Date

New creation date. The date is a packed 16-bit value with the following form:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (that is, add 1980 to get the actual year)

MilliSeconds

Number of 10 millisecond intervals in 2 seconds to add to the MS-DOS time. The number can be a value in the range 0 to 199.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 6Ch Group

Group

Opens or creates a file having the given name and attributes.

```
mov ah, 6Ch             ; Extended Open/Create
mov bx, ModeAndFlags    ; see below
mov cx, Attributes      ; see below
mov dx, Action          ; action to take
mov si, seg Filename    ; see below
mov ds, si
mov si, offset Filename
int 21h

jc error
mov [Handle], ax        ; file handle
mov [ActionTaken], cx   ; action taken to open file
```

Parameters

ModeAndFlags

Combination of access mode, sharing mode, and open flags. This parameter can be one value each from the access and sharing modes and any combination of open flags:

Access mode Meaning

OPEN_ACCESS_READONLY (0000h)

Opens the file for reading only.

OPEN_ACCESS_WRITEONLY (0001h)

Opens the file for writing only.

OPEN_ACCESS_READWRITE (0002h)

Opens the file for reading and writing.

0003h

Reserved; do not use.

OPEN_ACCESS_RO_NOMODLASTACCESS (0004h)

Opens the file for reading only without modifying the file's last access date.

Sharing mode Meaning

OPEN_SHARE_COMPATIBLE (0000h)

Opens the file with compatibility mode, allowing any process on a given computer to open the file any number of times.

OPEN_SHARE_DENYREADWRITE (0010h)

Opens the file and denies both read and write access to other processes.

OPEN_SHARE_DENYWRITE (0020h)

Opens the file and denies write access to other processes.

OPEN_SHARE_DENYREAD (0030h)

Opens the file and denies read access to other processes.

OPEN_SHARE_DENYNONE (0040h)

Opens the file without denying read or write access to other processes, but no process may open the file with

compatibility mode.

Open flags **Meaning**

OPEN_FLAGS_NOINHERIT (0080h)

If this flag is set, a child process created with Load and Execute Program (Interrupt 21h Function 4B00h) does not inherit the file handle. If the handle is needed by the child process, the parent process must pass the handle value to the child process. If this flag is not set, child processes inherit the file handle.

OPEN_FLAGS_NOCRITERR (2000h)

If a critical error occurs while MS-DOS is opening this file, Critical-Error Handler (Interrupt 24h) is not called. Instead, MS-DOS simply returns an error value to the program.

OPEN_FLAGS_COMMIT (4000h)

After each write operation, MS-DOS commits the file (flushes the contents of the cache buffer to disk).

Attributes

Attributes for files that are created or truncated. This parameter may be a combination of these values:

_A_NORMAL (0000h)

The file can be read from or written to. This value is valid only if used alone.

_A_RDONLY (0001h)

The file can be read from, but not written to.

_A_HIDDEN (0002h)

The file is hidden and does not appear in an ordinary directory listing.

_A_SYSTEM (0004h)

The file is part of the operating system or is used exclusively by it.

_A_VOLID (0008h)

The name specified by *Filename* is used as the volume label for the current medium.

_A_ARCH (0020h)

The file is an archive file. Applications use this value to mark files for backup or removal.

Action

Action to take it the file exists or does not exist. This parameter can be a combination of these values:

FILE_CREATE (0010h)	Creates a new file if it does not already exist or fails if the file already exists.
FILE_OPEN (0001h)	Opens the file. The function fails if the file does not exist.
FILE_TRUNCATE (0002h)	Opens the file and truncates it to zero length (replaces the existing file). The function fails if the file does not exist.

The only valid combinations are FILE_CREATE combined with FILE_OPEN or FILE_CREATE

combined with FILE_TRUNCATE.

Filename

Address of a null-terminated string specifying the name of the file to be opened or created. The name must be in the standard MS-DOS 8.3 filename format. The string must be a valid path for the volume associated with the given drive.

Return Value

Clears the carry flag, copies the file handle to the AX register, and sets CX to one of the following values if successful:

ACTION_OPENED (0001h)
ACTION_CREATED_OPENED (0002h)
ACTION_REPLACED_OPENED (0003h)

Otherwise, this function sets the carry flag and sets the AX register to one of the following error values:

ERROR_INVALID_FUNCTION (0001h)
ERROR_FILE_NOT_FOUND (0002h)
ERROR_PATH_NOT_FOUND (0003h)
ERROR_TOO_MANY_OPEN_FILES (0004h)
ERROR_ACCESS_DENIED (0005h)

Remarks

This function does not support long filenames. If the specified name is too long, this function truncates the name to the standard 8.3 format following the same naming scheme that the system uses when creating an alias for a long filename.

A file on a remote directory − that is, a directory on the network − cannot be opened, unless appropriate permissions for the directory exist.

Interrupt 21h Function 710Dh Group

Group

Flushes file system buffers and caches and optionally remounts the drivespace volume. Any write operations that the system has buffered are performed, and all waiting data is written to the appropriate drive.

```
mov ax, 710Dh      ; Reset Drive
mov cx, Flag       ; see below
mov dx, DriveNum   ; see below
int 21h

jc  error
```

Parameters

Flag

Flag specifying whether the system should flush and invalidate the data in the cache as well as the file system buffers. This parameter must be one of these values:

0000h	Resets the drive and flushes the file system buffers for the given drive.
0001h	Resets the drive, flushes the file system buffers, and flushes and invalidates the cache for the specified drive.
0002h	Remounts the drivespace volume.

The *Flag* value of 0002h is only supported on drivespace volumes. You should specify this value when the on-media format of the drivespace volume has changed and you want the file system to reinitialize and read the new format.

DriveNum

Drive to reset. This parameter can be 0 for the default drive, 1 for A, 2 for B, and so on.

Return Value

This function has no return value.

Interrupt 21h Function 7139h Group

Group

Creates a new directory having the given name.

```
mov ax, 7139h     ; Make Directory
mov dx, seg Name  ; see below
mov ds, dx
mov dx, offset Name
int 21h

jc  error
```

Parameters

Name

Address of a null-terminated string specifying the name of the directory to create. Long filenames are allowed.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 713Ah Group

Group

Removes the given directory. The directory must be empty.

```
mov ax, 713Ah     ; Remove Directory
mov dx, seg Name  ; see below
mov ds, dx
mov dx, offset Name
int 21h

jc  error
```

Parameters

Name

Address of a null-terminated string specifying the name of the directory to remove. Long filenames are allowed.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

The root directory cannot be deleted.

Interrupt 21h Function 713Bh Group

Group

Changes the current directory to the directory specified by the given path.

```
mov ax, 713Bh     ; Change Directory
mov dx, seg Path  ; see below
mov ds, dx
mov dx, offset Path
int 21h

jc error
```

Parameters

Path

Address of a null-terminated string specifying the directory to change to. The path, which can include the drive letter, must be a valid path for the given volume. Long filenames are allowed.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

The current or default directory is the directory that the system uses whenever an application supplies a filename that does not explicitly specify a directory. Similarly, the current or default drive is the drive the system uses whenever an application supplies a path that does not explicitly specify a drive. If a drive other than the default drive is specified as part of the new directory path, this function changes the current directory on that drive but does not change the default drive. Set Default Drive (Interrupt 21h Function 0Eh) can be used to change the default drive.

Interrupt 21h Function 7141h Group

Group

Deletes the given file or files. If the specified filename contains a wildcard character, this function can delete multiple files that match the wildcard.

```
mov ax, 7141h             ; Delete File
mov ch, MustMatchAttrs    ; see below
mov cl, SearchAttrs       ; see below
mov dx, seg Filename      ; see below
mov ds, dx
mov dx, offset Filename
mov si, WildcardAndAttrs  ; see below
int 21h

jc error
```

Parameters

MustMatchAttrs

Additional filter on the attributes specified in *SearchAttrs*. This parameter can be a combination of these values:

_A_NORMAL (0000h)

The file can be read from or written to. This value is valid only if used alone.

_A_RDONLY (0001h)

The file can be read from, but not written to.

_A_HIDDEN (0002h)

The file is hidden and does not appear in an ordinary directory listing.

_A_SYSTEM (0004h)

The file is part of the operating system or is used exclusively by it.

_A_VOLID (0008h)

The name specified by *Filename* is used as the volume label for the current medium.

_A_SUBDIR (0010h)

The name specified by *Filename* is used as a directory, not a file.

_A_ARCH (0020h)

The file is an archive file. Applications use this value to mark files for backup or removal.

SearchAttrs

File attributes to search for. This parameter can be a combination of these values:

_A_NORMAL (0000h)

_A_RDONLY (0001h)

_A_HIDDEN (0002h)

_A_SYSTEM (0004h)

_A_VOLID (0008h)

_A_SUBDIR (0010h)

_A_ARCH (0020h)

Filename

Address of a null-terminated string specifying the name of the file to delete. If *WildcardAndAttrs* is 1, the "*" and "?" wildcard characters are permitted in the filename. Long filenames are allowed.

WildcardAndAttrs

Search criteria. This parameter must be one of these values:

0 Wildcard characters are not allowed in *Filename*. Any specified attributes are ignored.

1 Wildcard characters are allowed in *Filename*. Files with specified attributes are matched.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

Wildcard searches are more flexible in Windows 95 than in MS-DOS. Both long filenames and aliases are considered in searches. For example, **1* finds Windows 95 filenames (both long filenames and aliases) that end in a *1* and **mid** finds filenames that contain the characters *mid*. In MS-DOS and in Windows 95 searching on real-mode FAT directories, all characters after the first * are ignored.

For more information about how *MustMatchAttrs* and *SearchAttrs* are used, see the comments for Find First File (Interrupt 21h Function 714Eh).

Interrupt 21h Function 7143h Group

Group

Retrieves or sets the file attributes, gets the compressed file size, or retrieves or sets the date and time for the given file.

```
mov ax, 7143h         ; Get or Set File Attributes
mov bl, Action        ; see below
mov cx, Attributes    ; see below
mov di, Date          ; see below
mov cx, Time          ; see below
mov si, MilliSeconds  ; see below
mov dx, seg Filename  ; see below
mov ds, dx
mov dx, offset Filename
int 21h

jc  error
; see below for return values
```

Parameters

Action

Action to take. This parameter can be one of the following values:

0	Retrieve attributes.
1	Set specified attributes.
2	Get physical size of a compressed file.
3	Set last write date/time.
4	Get last write date/time.
5	Set last access date.
6	Get last access date.
7	Set creation date/time.
8	Get creation date/time.

Attributes

File attributes to set, which are used only if *Action* is 1. This parameter can be a combination of these values:

_A_NORMAL (0000h)
: The file can be read from or written to. This value is valid only if used alone.

_A_RDONLY (0001h)
: The file can be read from, but not written to.

_A_HIDDEN (0002h)
: The file is hidden and does not appear in an ordinary directory listing.

_A_SYSTEM (0004h)
: The file is part of the operating system or is used exclusively by it.

_A_ARCH (0020h)
: The file is an archive file. Applications use this value to mark

files for backup or removal.

Time
New time to set, which is used only if *Action* is 3 (set last write date/time) or 7 (set creation date/time) The time is a packed 16-bit value with the following form:

Bits	Contents
0-4	Second divided by 2
5-10	Minute (0-59)
11-15	Hour (0-23 on a 24-hour clock)

Date
New date to set, which is used only if *Action* is 3 (set last write date/time), 5 (set last access date), or 7 (set creation date/time). The date is a packed 16-bit value with the following form:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (that is, add 1980 to get the actual year)

MilliSeconds
Number of 10 millisecond intervals in 2 seconds to add to the MS-DOS time. The number can be a value in the range 0 to 199. This value is only used if *Action* is 7 (set creation date/time).

Filename
Address of a null-terminated string specifying the name of the file to retrieve or set attributes for. Long filenames are allowed.

Return Value

Clears the carry flag if successful.

If *Action* is zero (retrieve attributes), the file attributes returned in the CX register may be a combination of the following values :

_A_NORMAL (0000h)
_A_RDONLY (0001h)
_A_HIDDEN (0002h)
_A_SYSTEM (0004h)
_A_VOLID (0008h)
_A_SUBDIR (0010h)
_A_ARCH (0020h)

If *Action* is 2 (get physical size of a compressed file), the size, in bytes, of the compressed file is returned in DX:AX. This value is the physical size of the compressed file – that is, the actual number of bytes that the compressed file occupies on disk.

If *Action* is 4 (get last write date/time) or 8 (get creation date/time), the CX register contains the time as a packed 16-bit value with the following form:

Bits	Contents
0-4	Second divided by 2
5-10	Minute (0-59)
11-15	Hour (0-23 on a 24-hour clock)

If *Action* is 4 (get last write date/time), 6 (get last access date), or 8 (get creation date/time), the DI register contains the date as a packed 16-bit value with the following form:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (that is, add 1980 to get the actual year)

If *Action* is 8 (get creation date/time), the SI register contains the number of 10 millisecond intervals in 2 seconds to add to the MS-DOS time. The number can be a value in the range of 0 to 199.

If the function is not successful, it sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 7147h Group

Group

Copies the path of the current directory for the given drive to the buffer. The copied path does not include the drive letter or the leading backslash.

```
mov ax, 7147h       ; Get Current Directory
mov dl, Drive       ; see below
mov si, seg Buffer  ; see below
mov ds, si
mov si, offset Buffer
int 21h

jc  error
```

Parameters

Drive

Drive number. This parameter can be 0 for current drive, 1 for A, 2 for B, and so on.

Buffer

Address of the buffer that receives the path. The buffer must be at least as big as the maximum allowed path for this volume as returned by Get Volume Information (Interrupt 21h Function 71A0h).

Return Value

Clears the carry flag and copies the path if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 714Eh Group

Group

Searches a directory for the first file or directory whose name and attributes match the specified name and attributes.

```
mov ax, 714Eh             ; Find First File
mov ch, MustMatchAttrs    ; see below
mov cl, SearchAttrs       ; see below
mov dx, seg Filename      ; see below
mov ds, dx
mov dx, offset Filename
mov di, seg FindData      ; see below
mov es, di
mov di, offset FindData
mov si, DateTimeFormat    ; see below
int 21h

jc  error
mov [Handle], ax          ; search handle
mov [ConversionCode], cx  ; Unicode to OEM/ANSI conversion OK?
```

Parameters

MustMatchAttrs

Additional filter on the attributes specified in *SearchAttrs*. This parameter can be a combination of these values:

_A_NORMAL (0000h)

The file can be read from or written to. This value is valid only if used alone.

_A_RDONLY (0001h)

The file can be read from, but not written to.

_A_HIDDEN (0002h)

The file is hidden and does not appear in an ordinary directory listing.

_A_SYSTEM (0004h)

The file is part of the operating system or is used exclusively by it.

_A_VOLID (0008h)

The name specified by *Filename* is used as the volume label for the current medium.

_A_SUBDIR (0010h)

The name specified by *Filename* is used as a directory, not a file.

_A_ARCH (0020h)

The file is an archive file. Applications use this value to mark files for backup or removal.

SearchAttrs

File attributes to search for. This parameter can be a combination of these values:

_A_NORMAL (0000h)

_A_RDONLY (0001h)
_A_HIDDEN (0002h)
_A_SYSTEM (0004h)
_A_VOLID (0008h)
_A_SUBDIR (0010h)
_A_ARCH (0020h)

Filename
Address of a null-terminated string specifying the name of the file or directory to search for. The name, which must be a valid filename or directory name, can include the "*" and "?" wildcard characters. Long filenames are allowed.

FindData
Address of a **WIN32_FIND_DATA** structure that receives information about the file.

DateTimeFormat
Date and time format to be returned. This parameter must be one of these values:

0	Returns the date and time in 64-bit file time format.
1	Returns the MS-DOS date and time values. MS-DOS date and time values are returned in the low doubleword of the **FILETIME** structure. Within the doubleword, the date is returned in the high-order word; the time is in the low-order word.

Return Value

Clears the carry flag, copies information about the file to the specified buffer, returns the search handle in the AX register, and sets the CX register to a combination of the following values if successful:

0x0000	All characters in the primary and alternate name members in the structure specified by *FindData* were successfully converted from Unicode.
0x0001	The primary name returned in the structure specified by *FindData* contains underscore characters in place of characters that could not be converted from Unicode.
0x0002	The alternate name returned in the structure specified by *FindData* contains underscore characters in place of characters that could not be converted from Unicode.

Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

Find First File and subsequent calls to Find Next File (Interrupt 21h Function 714Fh) use the following algorithm to match the attributes of a file or directory (referred to as *Attributes* in the algorithm) against *MustMatchAttrs* and *SearchAttrs*.

```
if  ((((<MustMatchAttrs> & ~<Attributes>) & 0x3F) == 0)
      && (((~<SearchAttrs> & <Attributes>) & 0x1E) == 0))
    {
         return the file or directory name
    }
    else
    {
         continue searching for the next name
    }
```

The following table lists the *MustMatchAttrs* and *SearchAttrs* values for some common searches where the specified filename is "*.*". In the table, the word normal means that the read only, hidden, or system attributes have not been set. Parentheses are used to indicate that a file or directory has more than one attribute. For example, (hidden and system) indicates that a file or directory has both the hidden attribute and the system attribute.

MustMatchAttrs	**SearchAttrs**	**Find results**
10h	10h	All normal directories
10h	12h	All normal and hidden directories
10h	14h	All normal and system directories
10h	16h	All normal, hidden, system and (hidden and system) directories
12h	12h	All hidden directories
14h	14h	All system directories
16h	16h	All (hidden and system) directories
00h	00h	All normal files
00h	01h	All normal and read only files
00h	02h	All normal and hidden files
00h	04h	All normal and system files
00h	06h	All normal, hidden, system, and (hidden and system) files
00h	10h	All normal files and directories
01h	01h	All read only files
02h	02h	All hidden files
02h	06h	All hidden and (hidden and system) files

This function can be used to return the volume label by specifying only _A_VOLID (0008h) in both *MustMatchAttrs* and *SearchAttrs.*

An application may use the handle returned in the AX register in subsequent calls to Find Next File (Interrupt 21h Function 714Fh). It is important to close the handle when it is no longer needed by calling Find Close (Interrupt 21h Function 71A1h).

Wildcard searches are more flexible in Windows 95 than in MS-DOS. For example, **1* finds the filenames (both long filenames and aliases) that end in a *1*, and **mid** finds filenames that contain the characters *mid*. In MS-DOS and in Windows 95 searching on real-mode FAT directories, all characters after the first * are ignored.

Interrupt 21h Function 714Fh Group

Group

Searches for the next file in a directory, returning information about the file in the given buffer.

```
mov ax, 714Fh            ; Find Next File
mov bx, Handle           ; see below
mov di, seg FindData     ; see below
mov es, di
mov di, offset FindData
mov si, DateTimeFormat   ; see below
int 21h

jc error
mov [ConversionCode], cx  ; Unicode to OEM/ANSI conversion OK?
```

Parameters

Handle

Search handle. It must have been previously returned from Find First File (Interrupt 21h Function 714Eh).

FindData

Address of a **WIN32_FIND_DATA** structure that receives information about the file.

DateTimeFormat

Date and time format to be returned. This parameter must be one of these values:

0	Returns the date and time in 64-bit file time format.
1	Returns the MS-DOS date and time values. MS-DOS date and time values are returned in the low doubleword of the **FILETIME** structure. Within the doubleword, the date is returned in the high-order word; the time is in the low-order word.

Return Value

Clears the carry flag, copies information to the specified buffer, and sets the CX register to a combination of these values if successful:

0x0000	All characters in the primary and alternate name member in the structure specified by *FindData* were successfully converted from Unicode.
0x0001	The primary name returned in the structure specified by *FindData* contains underscore characters in place of characters that could not be converted from Unicode.
0x0002	The alternate name returned in the structure specified by *FindData* contains underscore characters in place of characters that could not be converted from Unicode.

Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

It is important to close the handle when it is no longer needed by calling Find Close (Interrupt 21h Function 71A1h).

Interrupt 21h Function 7156h Group

Group

Changes the name of the given file or directory to the new name.

```
mov ax, 7156h           ; Rename File
mov dx, seg OldName     ; see below
mov ds, dx
mov dx, offset OldName
mov di, seg NewName     ; see below
mov es, di
mov di, offset NewName
int 21h

jc  error
```

Parameters

OldName

Address of a null-terminated string specifying the original name of the file or the directory to rename. Long filenames are allowed.

NewName

Address of a null-terminated string specifying the new name for the file or the directory. The function will fail if this parameter specifies an existing file or directory. The new name must not specify a drive different than the original drive. Long filenames are allowed.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Interrupt 21h Function 7160h Minor Code 0h

Group

Group

Retrieves the full path for the specified file or path.

```
mov ax, 7160h
mov cl, 0                   ; Get Full Path Name
mov ch, SubstExpand         ; see below
mov si, seg SourcePath      ; see below
mov ds, si
mov si, offset SourcePath
mov di, seg DestPath        ; see below
mov es, di
mov di, offset DestPath
int 21h

jc  error
```

Parameters

SubstExpand

Flag that indicates whether the returned path should contain a SUBST drive letter or the path associated with the SUBST drive. Zero is specified to indicate that the returned path should contain the path associated with the SUBST drive, and 80h is specified to indicate that the returned path should contain the SUBST drive letter.

SourcePath

Address of a null-terminated string that names the file or path to retrieve the full path for. Either the long filename or the standard 8.3 filename format is acceptable.

DestPath

Address of the buffer that receives the full path. The buffer should be large enough to contain the largest possible Windows 95 path (260 characters, including the drive letter, colon, leading backslash, and terminating null character).

Return Value

Clears the carry flag, modifies the AX register, and returns the full path in the given buffer if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

When just a filename is specified, this function merges the name of the current drive and directory with the specified filename to determine the full path. Relative paths containing the characters "." and ".." in *SourcePath* are fully expanded. The function does no validation, so the specified filename or path does not need to exist.

Interrupt 21h Function 7160h Minor Code 1h

Group

Group

Retrieves the complete path in its short form (the standard 8.3 format) for the specified file or path. The function returns the 8.3 filename for all directories in the path.

```
mov ax, 7160h
mov cl, 1                   ; Get Short Path Name
mov ch, SubstExpand         ; see below
mov si, seg SourcePath      ; see below
mov ds, si
mov si, offset SourcePath
mov di, seg DestPath        ; see below
mov es, di
mov di, offset DestPath
int 21h
jc  error
```

Parameters

SubstExpand

Flag that indicates if the returned path should contain a SUBST drive letter or the path associated with the SUBST drive. Zero is specified to indicate that the returned path should contain the path associated with the SUBST drive, and 80h is specified to indicate that the returned path should contain the SUBST drive letter.

SourcePath

Address of a null-terminated string that names the file or path to retrieve the complete short path for. Either the long or short form is acceptable as the source string.

DestPath

Address of the buffer that receives the complete path. The buffer should be large enough to contain the largest possible Windows 95 path in the short form (260 characters, including the drive letter, colon, leading backslash, and terminating null character).

Return Value

Clears the carry flag, modifies the AX register, and returns the complete short path in the given buffer if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

Relative paths containing the characters "." and ".." in *SourcePath* are fully expanded. Since this function performs validation, *SourcePath* must contain either a valid filename or path.

Interrupt 21h Function 7160h Minor Code 2h

Group

Group

Retrieves the complete path in its long filename form for the specified file or path. The function returns the long name for all directories in the path.

```
mov ax, 7160h
mov cl, 2                   ; Get Long Path Name
mov ch, SubstExpand         ; see below
mov si, seg SourcePath      ; see below
mov ds, si
mov si, offset SourcePath
mov di, seg DestPath        ; see below
mov es, di
mov di, offset DestPath
int 21h
jc  error
```

Parameters

SubstExpand

Flag that indicates if the returned path should contain a SUBST drive letter or the path associated with the SUBST drive. Zero is specified to indicate that the returned path should contain the path associated with the SUBST drive, and 80h is specified to indicate that the returned path should contain the SUBST drive letter.

SourcePath

Address of a null-terminated string that names the file or path to retrieve the complete long path for. Either the long filename or the short form is acceptable as the source string.

DestPath

Address of the buffer that receives the complete path. The buffer should be large enough to contain the largest possible Windows 95 path (260 characters, including the drive letter, colon, leading backslash, and terminating null character).

Return Value

Clears the carry flag, modifies the AX register, and returns the complete long path in the given buffer if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

Relative paths containing the characters "." and ".." in *SourcePath* are fully expanded. Since this function performs validation, *SourcePath* must contain either a valid filename or path.

Interrupt 21h Function 716Ch Group

Group

Creates or opens a file.

```
mov ax, 716Ch          ; Create or Open File
mov bx, ModeAndFlags   ; see below
mov cx, Attributes     ; see below
mov dx, Action         ; see below
mov si, seg Filename   ; see below
mov ds, si
mov si, offset Filename
mov di, AliasHint      ; see below
int 21h

jc error
mov [Handle], ax       ; file handle
mov [ActionTaken], cx  ; action taken to open file
```

Parameters

ModeAndFlags

Combination of access mode, sharing mode, and open flags. This parameter can be one value each from the access and sharing modes and any combination of open flags:

Access mode Meaning

OPEN_ACCESS_READONLY (0000h)

Opens the file for reading only.

OPEN_ACCESS_WRITEONLY (0001h)

Opens the file for writing only.

OPEN_ACCESS_READWRITE (0002h)

Opens the file for reading and writing.

0003h

Reserved; do not use.

OPEN_ACCESS_RO_NOMODLASTACCESS (0004h)

Opens the file for reading only without modifying the file's last access date.

Sharing mode Meaning

OPEN_SHARE_COMPATIBLE (0000h)

Opens the file with compatibility mode, allowing any process on a given computer to open the file any number of times.

OPEN_SHARE_DENYREADWRITE (0010h)

Opens the file and denies both read and write access to other processes.

OPEN_SHARE_DENYWRITE (0020h)

Opens the file and denies write access to other processes.

OPEN_SHARE_DENYREAD (0030h)

Opens the file and denies read access to other processes.

OPEN_SHARE_DENYNONE (0040h)

Opens the file without denying read or write access to other

processes, but no process may open the file for compatibility access.

Open flags **Meaning**

OPEN_FLAGS_NOINHERIT (0080h)

If this flag is set, a child process created with Load and Execute Program (Interrupt 21h Function 4B00h) does not inherit the file handle. If the handle is needed by the child process, the parent process must pass the handle value to the child process. If this flag is not set, child processes inherit the file handle.

OPEN_FLAGS_NO_BUFFERING (0100h)

The file is to be opened with no intermediate buffering or caching done by the system. Read and write operations access the disk directly. All reads and writes to the file must be done at file positions that are multiples of the disk sector size, in bytes, and the number of bytes read or written should also be a multiple of the sector size. Applications can determine the sector size, in bytes, with the Get Disk Free Space function (Interrupt 21h, Function 36h).

OPEN_FLAGS_NO_COMPRESS (0200h)

The file should not be compressed on a volume that performs file compression. If the volume does not perform file compression, this flag is ignored. This flag is valid only on file creation and is ignored on file open.

OPEN_FLAGS_ALIAS_HINT (0400h)

The number in the DI register is to be used as the numeric tail for the alias (short filename). For more information, see *AliasHint* below.

OPEN_FLAGS_NOCRITERR (2000h)

If a critical error occurs while MS-DOS is opening this file, Critical-Error Handler (Interrupt 24h) is not called. Instead, MS-DOS simply returns an error value to the program.

OPEN_FLAGS_COMMIT (4000h)

After each write operation, MS-DOS commits the file (flushes the contents of the cache buffer to disk).

Attributes

Attributes for files that are created or truncated. This parameter may be a combination of these values:

_A_NORMAL (0000h)

The file can be read from or written to. This value is valid only if used alone.

_A_RDONLY (0001h)

The file can be read from, but not written to.

_A_HIDDEN (0002h)

The file is hidden and does not appear in an ordinary directory listing.

_A_SYSTEM (0004h)

The file is part of the operating system or is used exclusively

by it.

_A_VOLID (0008h)

The name specified by *Filename* is used as the volume label for the current medium and is restricted to the standard 8.3 format. For information about an alternative way to set the volume label, see Set Media ID (Interrupt 21h Function 440Dh Minor Code 46h) in the *Microsoft MS-DOS Programmer's Reference*.

_A_ARCH (0020h)

The file is an archive file. Applications use this value to mark files for backup or removal.

Action

Action to take it the file exists or if it does not exist. This parameter can be a combination of these values:

FILE_CREATE (0010h)	Creates a new file if it does not already exist. The function fails if the file already exists.
FILE_OPEN (0001h)	Opens the file. The function fails if the file does not exist.
FILE_TRUNCATE (0002h)	Opens the file and truncates it to zero length (replaces the existing file). The function fails if the file does not exist.

The only valid combinations are FILE_CREATE combined with FILE_OPEN or FILE_CREATE combined with FILE_TRUNCATE.

Filename

Address of a null-terminated string specifying the name of the file to be opened or created. The string must be a valid path for the volume associated with the given drive. Long filenames are allowed.

AliasHint

Number that is used in the numeric tail for the alias (short filename). A numeric tail, which consists of the tilde character (~) followed a number, is appended to the end of a filename. The system constructs the alias from the first few characters of the long filename followed by the numeric tail. The system starts with the number 1 in the numeric tail. If that filename is in use, it uses the number 2. It continues in this fashion until a unique name is found. To override the default numbering scheme, you must specify the OPEN_FLAGS_ALIAS_HINT value when you create the file in addition to specifying this parameter. If a filename already exists with the specified numeric tail, the system uses the default numbering scheme. You should specify a number for this parameter, not the tilde character.

Return Value

Clears carry flag, copies the file handle to the AX register, and sets the CX register to one of the following values if successful:

ACTION_OPENED (0001h)
ACTION_CREATED_OPENED (0002h)
ACTION_REPLACED_OPENED (0003h)

Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

A file on a remote directory − that is, a directory on the network − cannot be opened, created, or modified, unless the appropriate permissions for the directory exist.

Interrupt 21h Function 71A0h Group

Group

Returns information about the volume associated with the given root directory.

```
mov ax, 71A0h          ; Get Volume Information
mov di, seg Buffer     ; see below
mov es, di
mov di, offset Buffer
mov cx, BufSize        ; see below
mov dx, seg RootName   ; see below
mov ds, dx
mov dx, offset RootName
int 21h

jc  error
mov [Flags], bx        ; file system flags
mov [MaxFilename], cx  ; max. filename length, excluding null
mov [MaxPath], dx      ; max. path length, including null
```

Parameters

Buffer

Address of a buffer that receives a null-terminated string specifying the name of the file system.

BufSize

Size, in bytes, of the buffer that receives the name. The buffer should include space for the terminating null character.

RootName

Address of a null-terminated string specifying the name of the root directory of the volume to check. This parameter must not be NULL, or the function will fail. The format for this parameter is "C:\".

Return Value

Clears the carry flag, copies the file system name to the buffer given by the ES:DI register pair, and sets the BX, CX, and DX registers to the following values if successful:

BX	File system flags, which can be a combination of these values: FS_CASE_SENSITIVE (0001h) Specifies that searches are case-sensitive. FS_CASE_IS_PRESERVED (0002h) Preserves case in directory entries. FS_UNICODE_ON_DISK (0004h) Uses Unicode characters in file and directory names. FS_LFN_APIS (4000h) Supports new long filename functions. FS_VOLUME_COMPRES

SED (8000h)
Specifies that the volume is compressed.

CX	Maximum allowed length, excluding the terminating null character, of a filename for this volume. For example, on the protected-mode FAT file system, this value is 255.
DX	Maximum allowed length of a path for this volume, including the drive letter, colon, leading slash, and terminating null character. For example, on the protected-mode FAT file system, this value is 260.

Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

This function accesses the disk the first time it is called, but subsequent calls do not access the disk.

Interrupt 21h Function 71A1h Group

Group

Closes the file search identified by the search handle.

```
mov ax, 71A1h   ; Find Close
mov bx, Handle  ; see below
int 21h

jc  error
```

Parameters

Handle

Search handle. It must have been previously returned from Find First File (Interrupt 21h Function 714Eh).

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

Unlike MS-DOS Find First File (Interrupt 21h Function 4Eh), the long filename version of Find First File (Interrupt 21h Function 714Eh) allocates internal storage for the search operations and returns a handle that identifies the storage. This handle is used with Find Next File. To make sure this internal storage is freed, you must call Find Close to end the search.

Interrupt 21h Function 71A6h Group

Group

Retrieves information about the specified file.

```
mov ax, 71a6h              ; Get File Info By Handle
mov bx, Handle             ; see below
mov dx, seg lpFileInfo     ; see below
mov ds, dx
mov dx, offset lpFileInfo
stc                        ; must set carry flag
int 21h

jnc success
cmp ax, 7100h
je not_supported
```

Parameters

Handle

File handle to retrieve information about.

lpFileInfo

Address of a **BY_HANDLE_FILE_INFORMATION** structure that receives the file information. The structure can be used in subsequent calls to Get File Info By Handle to refer to the information about the file.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

Note that it is important to explicitly set the carry flag before calling this function.

Interrupt 21h Function 71A7h Minor Code 0h

Group

Group

Converts a 64-bit file time to MS-DOS date and time values.

```
mov ax, 71A7h           ; date and time format conversion
mov bl, 0               ; File Time To DOS Time
mov si, seg lpft        ; see below
mov ds, si
mov si, offset lpft
int 21h

jc  error
mov [DOSTime], cx
mov [DOSDate], dx
mov [MilliSeconds], bh  ; number of 10ms intervals in 2 seconds
```

Parameters

lpft

Address of a **FILETIME** structure containing the 64-bit file time to convert to the MS-DOS date and time format.

Return Value

Clears the carry flag, and sets the BH, CX, and DX registers to these values if successful:

BH — Number of 10 millisecond intervals in 2 seconds to add to the MS-DOS time. It can be a value in the range 0 to 199.

CX — MS-DOS time. The time is a packed 16-bit value with the following form:

Bits	Contents
0-4	Second divided by 2
5-10	Minute (0-59)
11-15	Hour (0-23 on a 24-hour clock)

DX — MS-DOS date. The date is a packed 16-bit value with the following form:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (that is, add 1980 to get the actual year)

Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

The MS-DOS date format can represent only dates between 1/1/1980 and 12/31/2107; this conversion fails if the input file time is outside this range.

The time in **FILETIME** must be Coordinated Universal Time (UTC). The MS-DOS time is local time.

Interrupt 21h Function 71A7h Minor Code 1h

Group

Group

Converts MS-DOS date and time values to 64-bit file time.

```
mov ax, 71A7h         ; date and time format conversion
mov bl, 1             ; Dos Time To File Time
mov bh, MilliSeconds  ; see below
mov cx, DOSTime       ; see below
mov dx, DOSDate       ; see below
mov di, seg lpft      ; see below
mov es, di
mov di, offset lpft
int 21h

jc  error
```

Parameters

MilliSeconds

Number of 10 millisecond intervals in 2 seconds to add to the MS-DOS time. The number can be a value in the range 0 to 199.

DOSTime

MS-DOS time to convert. The time is a packed 16-bit value with the following form:

Bits	Contents
0-4	Second divided by 2
5-10	Minute (0-59)
11-15	Hour (0-23 on a 24-hour clock)

DOSDate

MS-DOS date to convert. The date is a packed 16-bit value with the following form:

Bits	Contents
0-4	Day of the month (1-31)
5-8	Month (1 = January, 2 = February, and so on)
9-15	Year offset from 1980 (that is, add 1980 to get the actual year)

lpft

Address of a **FILETIME** structure to receive the converted 64-bit file time.

Return Value

Clears the carry flag and returns the 64-bit file time in the specified structure if successful. Otherwise, the function sets the carry flag and sets the AX register to an error value.

Remarks

The time in **FILETIME** must be Coordinated Universal Time (UTC). The MS-DOS time is local time.

Interrupt 21h Function 71A8h Group

Group

Generates an alias (a filename in the 8.3 format) for the specified long filename.

```
mov ax, 71a8h               ; generate short name
mov si, seg LongFilename    ; see below
mov ds, si
mov si, offset LongFilename
mov di, seg ShortFilename   ; see below
mov es, di
mov di, offset ShortFilename
mov dh, ShortNameFormat     ; see below
mov dl, CharSet             ; see below
int 21h
```

Parameters

LongFilename
Address of null-terminated string that names the long filename to generate an alias for. This string must contain only a filename, not a path.

ShortFilename
Address of null-terminated string that receives the generated alias.

ShortNameFormat
Format for the returned alias (0 is specified for an 11 character directory entry format or 1 for an 8.3 format).

CharSet
Character set of both the long filename and alias. This parameter is a packed 8-bit value with the following form:

Bits	Contents
0-3	Specifies the character set of the source long filename.
4-7	Specifies the character set of the destination alias.

One of the following values is specified to indicate the character set for the long filename and alias:

BCS_WANSI (0)	Windows ANSI character set
BCS_OEM (1)	Current OEM character set
BCS_UNICODE (2)	Unicode character set

Return Value

Returns the generated alias in the specified buffer if successful.

Remarks

This function generates the alias using the same algorithm that the system uses with the exception that the returned alias will never have a numeric tail. A numeric tail is appended to the end of an alias and consists of the tilde character (~) followed a number. When the system generates an alias, it may attach a numeric tail if the default alias already exists in the current directory. Because this function does not check the current directory to see if the alias already exists, the returned alias will never have a numeric tail. This service is useful for disk utilities that are trying to establish whether the alias, which seems to be associated with a long filename, is correctly associated.

Interrupt 21h Function 71A9h Group

Group

Creates or opens a file. This function is for use by real-mode servers only. It takes the same parameters as Create or Open File (Interrupt 21h Function 716Ch) and returns a global file handle.

Interrupt 21h Function 71AAh Minor Code 0h

Group

Group

Associates a path with a drive letter.

```
mov ax, 71aah            ; SUBST
mov bh, 0                ; Create Subst
mov bl, DriveNum         ; see below
mov dx, seg PathName     ; see below
mov ds, dx
mov dx, offset PathName
int 21h

jc error
```

Parameters

DriveNum

Drive to SUBST. This parameter can be 0 for the default drive, 1 for A, 2 for B, and so on.

PathName

Address of path to associate the drive with.

Return Value

Clears the carry flag if successful. Otherwise the function sets the carry flag and returns an error value in the AX register

Interrupt 21h Function 71AAh Minor Code 1h

Group

Group

Terminate the association between a path and a drive letter.

```
mov ax, 71aah            ; SUBST
mov bh, 1                ; Terminate Subst
mov bl, DriveNum         ; see below
int 21h

jc error
```

Parameters

DriveNum

Drive to terminate SUBST. This parameter can 1 for A, 2 for B, and so on. Note that *DriveNum* cannot be 0 to indicate the default drive.

Return Value

Clears the carry flag if successful. Otherwise, the function sets the carry flag and returns an error value in the AX register.

Interrupt 21h Function 71AAh Minor Code 2h

Group

Group

Determines if the specified drive is associated with a path and, if it is, retrieves the associated path.

```
mov ax, 71aah            ; SUBST
mov bh, 2                ; Query SUBST
mov bl, DriveNum         ; see below
mov dx, seg PathName     ; see below
mov ds, dx
mov dx, offset PathName
int 21h

jc error
```

Parameters

DriveNum

Drive to SUBST. This parameter can 1 for A, 2 for B, and so on. Note that *DriveNum* cannot be 0 to indicate the default drive.

PathName

Address of buffer that receives the null-terminated string of the path associated with the specified drive. The buffer must be of MAXPATHLEN size.

Return Value

Clears the carry flag and retrieves the associated path in the specified buffer if successful. Otherwise, the function sets the carry flag and returns an error value in the AX register.

Module32First Group

Group

Retrieves information about the first module associated with a process.

```
BOOL WINAPI Module32First(HANDLE hSnapshot, LPMODULEENTRY32 lpme);
```

Parameters

hSnapshot

Handle of the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lpme

Address of a buffer containing a **MODULEENTRY32** structure.

Return Value

Returns TRUE if the first entry of the module list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no modules exist or the snapshot does not contain module information.

Remarks

The calling application must set the **dwSize** member of **MODULEENTRY32** to the size, in bytes, of the structure. **Module32First** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To retrieve information about other modules associated with the specified process, use the **Module32Next** function.

Module32Next Group

Group

Retrieves information about the next module associated with a process or thread.

```
BOOL WINAPI Module32Next(HANDLE hSnapshot, LPMODULEENTRY32 lpme);
```

Parameters

hSnapshot

Handle of the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lpme

Address of a buffer containing a **MODULEENTRY32** structure.

Return Value

Returns TRUE if the next entry of the module list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no more modules exist.

Remarks

To retrieve information about first module associated with a process, use the **Module32First** function.

Process32First Group

Group

Retrieves information about the first process encountered in a system snapshot.

```
BOOL WINAPI Process32First(HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
```

Parameters

hSnapshot

Handle of the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lppe

Address of a **PROCESSENTRY32** structure.

Return Value

Returns TRUE if the first entry of the process list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no processes exist or the snapshot does not contain process information.

Remarks

The calling application must set the **dwSize** member of **PROCESSENTRY32** to the size, in bytes, of the structure. **Process32First** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To retrieve information about other processes recorded in the same snapshot, use the **Process32Next** function.

Process32Next Group

Group

Retrieves information about the next process recorded in a system snapshot.

```
BOOL WINAPI Process32Next(HANDLE hSnapshot, LPPROCESSENTRY32 lppe);
```

Parameters

hSnapshot

Handle of the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lppe

Address of a **PROCESSENTRY32** structure.

Return Value

Returns TRUE if the next entry of the process list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no processes exist or the snapshot does not contain process information.

Remarks

To retrieve information about the first process recorded in a snapshot, use the **Process32First** function.

Query Close Group

Group

Indicates whether the user has attempted to close an MS-DOS - based application from Windows by choosing the Close command from the system menu.

```
mov ah, 16h   ; Windows multiplex function
mov al, 8Fh   ; VM Close
mov dh, 1     ; Query Close
mov dl, 0     ; always 0
int 2Fh
```

Return Value

Returns one of the following values in the AX register:

0	The close command was chosen, and the application has not acknowledged the close state.
1	The close command was chosen, and the application has acknowledged the close state.
168Fh	The close command has not been chosen, and the application should continue running.

Remarks

Query Close returns a nonzero value if the application has not enabled the Close command by using the Enable or Disable Close Command function.

RegisterServiceProcess Group

Group

The **RegisterServiceProcess** function registers or unregisters a service process. A service process continues to run after the user logs off.

```
DWORD RegisterServiceProcess(DWORD dwProcessId,
    DWORD dwType);
```

Parameters

dwProcessId

Specifies the identifier of the process to register as a service process. Specifies NULL to register the current process.

dwType

Specifies whether the service is to be registered or unregistered. This parameter can be one of the following values.

Value	Meaning
RSP_SIMPLE_SERVICE	Registers the process as a service process.
RSP_UNREGISTER_SERVICE	Unregisters the process as a service process.

Return Value

The return value is 1 if successful or 0 if an error occurs.

Set Application Title Group

Group

Sets the application title to the given string.

```
mov  ah, 16h           ; Windows multiplex function
mov  al, 8Eh           ; VM Title
mov  di, seg AppTitle  ; see below
mov  es, di
mov  di, offset AppTitle
mov  dx, 0             ; Set Application Title
int  2Fh
cmp   ax, 1
je   success
```

Parameters

AppTitle

Address of a null-terminated string specifying the application title. The title must not exceed 80 characters, including the terminating null character. If this parameter is zero or points to an empty string, the function removes the current application title.

Return Value

Returns 1 in the AX register if successful or zero otherwise.

Remarks

Although not common, Set Application Title may return 1 in the AX register even though the title was not changed. In general, applications must not rely on the operating system to keep an accurate copy of the current title.

Set Virtual Machine Title Group

Group

Sets the virtual machine title to the given string. Applications should not change the virtual machine title except under explicit instructions from the user.

```
mov  ah, 16h           ; Windows multiplex function
mov  al, 8Eh           ; VM Title
mov  di, seg VMTitle   ; see below
mov  es, di
mov  di, offset VMTitle
mov  dx, 1             ; Set Virtual Machine Title
int  2Fh
cmp   ax, 1
je   success
```

Parameters

VMTitle

Address of a null-terminated string specifying the virtual machine title. The title must not exceed 30 characters, including the terminating null character. If this parameter is zero or points to an empty string, the function removes the current virtual machine title.

Return Value

Returns 1 in the AX register if successful or zero otherwise.

Remarks

Although not common, Set Virtual Machine Title may return 1 in the AX register even though the title was not changed. In general, applications must not rely on the operating system to keep an accurate copy of the current title.

Thread32First Group

Group

Retrieves information about the first thread of any process encountered in a system snapshot.

```
BOOL WINAPI Thread32First(HANDLE hSnapshot, LPTHREADENTRY32 lpte);
```

Parameters

hSnapshot

Handle of the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lpte

Address of a **THREADENTRY32** structure.

Return Value

Returns TRUE if the first entry of the thread list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no threads exist or the snapshot does not contain thread information.

Remarks

The calling application must set the **dwSize** member of **THREADENTRY32** to the size, in bytes, of the structure. **Thread32First** changes **dwSize** to the number of bytes written to the structure. This will never be greater than the initial value of **dwSize**, but it may be smaller. If the value is smaller, do not rely on the values of any members whose offsets are greater than this value.

To retrieve information about other threads recorded in the same snapshot, use the **Thread32Next** function.

Thread32Next Group

Group

Retrieves information about the next thread of any process encountered in the system memory snapshot.

```
BOOL WINAPI Thread32Next(HANDLE hSnapshot, LPTHREADENTRY32 lpte);
```

Parameters

hSnapshot

Handle of the snapshot returned from a previous call to the **CreateToolhelp32Snapshot** function.

lpte

Address of a **THREADENTRY32** structure.

Return Value

Returns TRUE if the next entry of the thread list has been copied to the buffer or FALSE otherwise. The ERROR_NO_MORE_FILES error value is returned by the **GetLastError** function if no threads exist or the snapshot does not contain thread information.

Remarks

To retrieve information about the first thread recorded in a snapshot, use the **Thread32First** function.

Toolhelp32ReadProcessMemory Group

Group

Copies memory allocated to another process into an application-supplied buffer.

```
BOOL WINAPI Toolhelp32ReadProcessMemory(DWORD th32ProcessID,
    LPCVOID lpBaseAddress, LPVOID lpBuffer, DWORD cbRead,
    LPDWORD lpNumberOfBytesRead);
```

Parameters

th32ProcessID
: Identifier of the process whose memory is being copied. This parameter can be zero to copy the memory of the current process.

lpBaseAddress
: Base address in the specified process to read. Before transferring any data, the system verifies that all data in the base address and memory of the specified size is accessible for read access. If this is the case, the function proceeds. Otherwise, the function fails.

lpBuffer
: Address of the buffer that receives the contents of the address space of the specified process.

cbRead
: Number of bytes to read from the specified process.

lpNumberOfBytesRead
: Number of bytes copied to the specified buffer. If this parameter is NULL, it is ignored.

Return Value

Returns TRUE if successful.

DIOC_REGISTERS Group

Group

Contains the register values for calling Interrupt 21h commands through the **DeviceIoControl** function. The meaning of the registers depends on the given command.

```
typedef struct DIOCRegs {
    DWORD   reg_EBX;
    DWORD   reg_EDX;
    DWORD   reg_ECX;
    DWORD   reg_EAX;
    DWORD   reg_EDI;
    DWORD   reg_ESI;
    DWORD   reg_Flags;
} DIOC_REGISTERS;
```

Members

reg_EBX
EBX register.

reg_EDX
EDX register.

reg_ECX
ECX register.

reg_EAX
EAX register

reg_EDI
EDI register.

reg_ESI
ESI register.

reg_Flags
Flags register.

Remarks

Some interrupt functions require far pointers passed in segment:offset pairs where the segment is placed in a segment register. Since the 32-bit code does not have segments, the **DIOC_REGISTERS** structure contains no segment registers. You should place the full pointer into the structure member that corresponds to the register used to hold the offset portion of the real-mode pointer. For example, use **reg_EDX** for pointers that go into the DS:DX registers.

DIOCParams Group

Group

Contains information that an application has passed to the virtual device driver (VxD) by calling the **DeviceIoControl** function.

```
include vwin32.inc

DIOCParams STRUC
    Internal1             DD  ?
    VMHandle              DD  ?
    Internal2             DD  ?
    dwIoControlCode       DD  ?
    lpvInBuffer           DD  ?
    cbInBuffer            DD  ?
    lpvOutBuffer          DD  ?
    cbOutBuffer           DD  ?
    lpcbBytesReturned     DD  ?
    lpoOverlapped         DD  ?
    hDevice               DD  ?
    tagProcess            DD  ?
DIOCParams ENDS
```

Members

Internal1
Reserved.

VMHandle
Handle of virtual machine.

Internal2
Reserved.

dwIoControlCode
Control code to process. This member can be a developer-defined value or one of these system-defined values:

DIOC_CLOSEHANDLE	Notifies a VxD that an application has closed its device handle for the VxD. The VxD should perform cleanup operations and release any structures associated with the application.
DIOC_GETVERSION	Queries the VxD to determine if it supports the device IOCTL interface. If the VxD supports the interface, it must clear the EAX register. Otherwise, it must place a nonzero value in EAX. If **cbOutBuffer** is nonzero and the VxD supports the interface, the VxD should copy version information to the **lpvOutBuffer**.

lpvInBuffer
Address of a buffer that contains data needed to process the control code.

cbInBuffer
Size, in bytes, of the buffer pointed to by **lpvInBuffer**.

lpvOutBuffer

Address of a buffer that receives the results of processing the control code.

cbOutBuffer

Size, in bytes, of the buffer pointed to by **lpvOutBuffer**.

lpcbBytesReturned

Number of bytes copied to the buffer pointed to by **lpvOutBuffer**.

lpoOverlapped

Address of an **OVERLAPPED** structure that contains information used to complete the command asynchronously. If the command is to be completed synchronously, this member is NULL.

hDevice

Handle of device.

tagProcess

Information that the VxD can use to tag the current request along with **hDevice**. When the VxD receives a DIOC_CLOSEHANDLE control code for **hDevice** and **tagProcess**, it should perform appropriate cleanup operations.

DRIVE_MAP_INFO Group

Group

Contains information about the drive specified in the call to Get Drive Map Info (Interrupt 2lh Function 440Dh Minor Code 6Fh).

```
DRIVE_MAP_INFO   struc
    dmiAllocationLength   db ?
    dmiInfoLength         db ?
    dmiFlags              db ?
    dmiInt13Unit          db ?
    dmiAssociatedDriveMap dd ?
    dmiPartitionStartRBA  dq ?
DRIVE_MAP_INFO   ends
```

Members

dmiAllocationLength

Length of the buffer provided by the application calling Get Drive Map Info. This value should be the size of the **DRIVE_MAP_INFO** structure.

dmiInfoLength

Number of bytes that Get Drive Map Info used in the buffer provided by the calling application. Typically, this value is the size of the **DRIVE_MAP_INFO** structure.

dmiFlags

Flags describing the given drive. This member, which is filled by Get Drive Map Info, can be a combination of these values:

PROT_MODE_LOGICAL_DRIVE (01h)	A protected-mode driver is in use for this logical drive.
PROT_MODE_PHYSICAL_DRIVE (02h)	A protected-mode driver is in use for the physical drive corresponding to this logical drive.
PROT_MODE_ONLY_DRIVE (04h)	The drive is not available when running with MS-DOS.
PROT_MODE_EJECT (08h)	A protected-mode drive supports an electronic eject operation.
PROT_MODE_ASYNC_NOTIFY (10h)	The drive issues media arrival and removal notifications. This value is currently used for CD-ROM drives that are controlled by the protected-mode driver and that cause a broadcast message when media is removed or inserted without the application having to make a request to the drive. It can also be used by disk drivers.

dmiInt13Unit

Physical drive number of the given drive. This member, which is filled by Get Drive Map Info, can be one of these values:

00 - 7Fh	Floppy disk drive (00 for the first floppy drive, 01 for

	the second, and so on).
80 - FEh	Hard disk drive (80 for the first hard disk drive, 81 for the second, and so on).
FFh	The given drive does not map to a physical drive.

dmiAssociatedDriveMap

Logical drive numbers that are associated with the given physical drive. For example, a host drive C with child drive letters A and B would return with bits 0 and 1 set.

dmiPartitionStartRBA

Relative block address offset from the start of the physical volume to the start of the given partition.

Remarks

Before an application makes a call to Get Drive Map Info, the **dmiAllocationLength** member must be set to the size of the **DRIVE_MAP_INFO** structure. All other members of the structure are filled in by Get Drive Map Info.

HEAPENTRY32 Group

Group

Describes one entry (block) of a heap that is being examined.

```
typedef struct tagHEAPENTRY32
{
    DWORD  dwSize;
    HANDLE hHandle;
    DWORD  dwAddress;
    DWORD  dwBlockSize;
    DWORD  dwFlags;
    DWORD  dwLockCount;
    DWORD  dwResvd;
    DWORD  th32ProcessID;
    DWORD  th32HeapID;
} HEAPENTRY32;
typedef HEAPENTRY32 *  PHEAPENTRY32;
typedef HEAPENTRY32 *  LPHEAPENTRY32;
```

Members

dwSize
Specifies the length, in bytes, of the structure. Before calling the **Heap32First** function, set this member to

sizeof(HEAPENTRY32).

hHandle
Handle of the heap block.

dwAddress
Linear address of the start of the block.

dwBlockSize
Size, in bytes, of the heap block.

dwFlags
Flags. These values are defined:

LF32_FIXED	The memory block has a fixed (unmovable) location.
LF32_FREE	The memory block is not used.
LF32_MOVEABLE	The memory block location can be moved.

dwLockCount
Lock count on the memory block. The lock count is incremented each time the **GlobalLock** or **LocalLock** function is called on the block either by the application or the DLL that the heap belongs to.

dwResvd
Reserved; do not use.

th32ProcessID
Identifier of the process to examine. The contents of this member can be used by Win32 API elements.

th32HeapID
Heap identifier in the owning process context. The contents of this member has meaning only to the tool help functions. It is not a handle, nor is it usable by Win32 API elements.

HEAPLIST32 Group

Group

Describes an entry from a list that enumerates the heaps used by a specified process.

```
typedef struct tagHEAPLIST32 {
    DWORD dwSize;
    DWORD th32ProcessID;
    DWORD th32HeapID;
    DWORD dwFlags;
} HEAPLIST32;
typedef HEAPLIST32 *  PHEAPLIST32;
typedef HEAPLIST32 *  LPHEAPLIST32;
```

Members

dwSize

Specifies the length, in bytes, of the structure. Before calling the **Heap32ListFirst** function, set this member to

sizeof(HEAPLIST32).

th32ProcessID

Identifier of the process to examine. The contents of this member can be used by Win32 API elements.

th32HeapID

Heap identifier in the owning process context. The contents of this member has meaning only to the tool help functions. It is not a handle, nor is it usable by Win32 API elements.

dwFlags

Flags. These values are defined:

HF32_DEFAULT	Process's default heap

MID Group

Group

Contains information that uniquely identifies a disk or other storage medium.

```
MID struc
    midInfoLevel   dw 0
    midSerialNum   dd ?
    midVolLabel    db 11 dup (?)
    midFileSysType db 8 dup (?)
MID ends
```

Members

midInfoLevel
Information level. This member must be zero.

midSerialNum
Serial number for the medium.

midVolLabel
Volume label for the medium. If the label has fewer than 11 characters, space characters (ASCII 20h) fill the remaining bytes in this member.

midFileSysType
Type of file system as an 8-byte ASCII string. This member can be one of these values:

FAT12	12-bit file allocation table (FAT)
FAT16	16-bit FAT
CDROM	High Sierra file system
CD001	ISO9660 file system
CDAUDIO	Audio disk

If the name has fewer than eight characters, space characters (ASCII 20h) fill the remaining bytes in this member.

MODULEENTRY32 Group

Group

Describes an entry from a list that enumerates the modules used by a specified process.

```
typedef struct tagMODULEENTRY32 {
    DWORD   dwSize;
    DWORD   th32ModuleID;
    DWORD   th32ProcessID;
    DWORD   GlblcntUsage;
    DWORD   ProccntUsage;
    BYTE  * modBaseAddr;
    DWORD   modBaseSize;
    HMODULE hModule;
    char    szModule[MAX_MODULE_NAME32 + 1];
    char    szExePath[MAX_PATH];
} MODULEENTRY32;
typedef MODULEENTRY32 *  PMODULEENTRY32;
typedef MODULEENTRY32 *  LPMODULEENTRY32;
```

Members

dwSize
Specifies the length, in bytes, of the structure. Before calling the **Module32First** function, set this member to

sizeof(MODULEENTRY32).

th32ModuleID
Module identifier in the context of the owning process. The contents of this member has meaning only to the tool help functions. It is not a handle, nor is it usable by Win32 API elements.

th32ProcessID
Identifier of the process being examined. The contents of this member can be used by Win32 API elements.

GlblcntUsage
Global usage count on the module.

ProccntUsage
Module usage count in the context of the owning process.

modBaseAddr
Base address of the module in the context of the owning process.

modBaseSize
Size, in bytes, of the module.

hModule
Handle of the module in the context of the owning process.

szModule
String containing the module name.

szExePath
String containing the location (path) of the module.

Note **modBaseAddr** and **hModule** are valid *only* in the context of the process specified by **th32ProcessID**.

PARAMBLOCK Group

Group

Contains information about locked drives.

```
PARAMBLOCK struc
    Operation db ?
    NumLocks  db ?
PARAMBLOCK ends
```

Members

Operation

Operation to carry out provided by the calling application of Lock/Unlock Removable Media (Interrupt 21h Function 440Dh Minor Code 48h). This member can be one of these values:

0	Locks the volume in the drive.
1	Unlocks the volume in the drive.
2	Returns the lock or unlock status.

All other values are reserved.

NumLocks

Number of locks pending on the given drive filled in by Lock/Unlock Removable Media.

PROCESSENTRY32 Group

Group

Describes an entry from a list that enumerates the processes residing in the system address space when a snapshot was taken.

```
typedef struct tagPROCESSENTRY32 {
    DWORD dwSize;
    DWORD cntUsage;
    DWORD th32ProcessID;
    DWORD th32DefaultHeapID;
    DWORD th32ModuleID;
    DWORD cntThreads;
    DWORD th32ParentProcessID;
    LONG  pcPriClassBase;
    DWORD dwFlags;
    char szExeFile[MAX_PATH];
} PROCESSENTRY32;
typedef PROCESSENTRY32 *  PPROCESSENTRY32;
typedef PROCESSENTRY32 *  LPPROCESSENTRY32;
```

Members

dwSize
Specifies the length, in bytes, of the structure. Before calling the **Process32First** function, set this member to

sizeof(PROCESSENTRY32).

cntUsage
Number of references to the process. A process exists as long as its usage count is nonzero. As soon as its usage count becomes zero, a process terminates.

th32ProcessID
Identifier of the process. The contents of this member can be used by Win32 API elements.

th32DefaultHeapID
Identifier of the default heap for the process. The contents of this member has meaning only to the tool help functions. It is not a handle, nor is it usable by Win32 API elements.

th32ModuleID
Module identifier of the process. The contents of this member has meaning only to the tool help functions. It is not a handle, nor is it usable by Win32 API elements.

cntThreads
Number of execution threads started by the process.

th32ParentProcessID
Identifier of the process that created the process being examined. The contents of this member can be used by Win32 API elements.

pcPriClassBase
Base priority of any threads created by this process.

dwFlags
Reserved; do not use.

szExeFile
Path and filename of the executable file for the process.

THREADENTRY32 Group

Group

Describes an entry from a list that enumerates the threads executing in the system when a snapshot was taken.

```
typedef struct tagTHREADENTRY32{
    DWORD  dwSize;
    DWORD  cntUsage;
    DWORD  th32ThreadID;
    DWORD  th32OwnerProcessID;
    LONG   tpBasePri;
    LONG   tpDeltaPri;
    DWORD  dwFlags;
} THREADENTRY32;
typedef THREADENTRY32 *  PTHREADENTRY32;
typedef THREADENTRY32 *  LPTHREADENTRY32;
```

Members

dwSize
Specifies the length, in bytes, of the structure. Before calling the **Thread32First** function, set this member to

sizeof(THREADENTRY32).

cntUsage
Number of references to the thread. A thread exists as long as its usage count is nonzero. As soon as its usage count becomes zero, a thread terminates.

th32ThreadID
Identifier of the thread. This identifier is compatible with the thread identifier returned by the **CreateProcess** function.

th32OwnerProcessID
Identifier of the process that created the thread. The contents of this member can be used by Win32 API elements.

tpBasePri
Initial priority level assigned to a thread. These values are defined:

THREAD_PRIORITY_IDLE
Indicates a base priority level of 1 for IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 16 for REALTIME_PRIORITY_CLASS processes.

THREAD_PRIORITY_LOWEST
Indicates 2 points below normal priority for the priority class.

THREAD_PRIORITY_BELOW_NORMAL
Indicates 1 point below normal priority for the priority class.

THREAD_PRIORITY_NORMAL
Indicates normal priority for the priority class.

THREAD_PRIORITY_ABOVE_NORMAL
Indicates 1 point above normal priority for the priority class.

THREAD_PRIORITY_HIGHEST
Indicates 2 points above normal priority for the priority class.

THREAD_PRIORITY_TIME_CRITICAL
Indicates a base priority level of 15 for IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 31 for

REALTIME_PRIORITY_CLASS processes.

tpDeltaPri

Change in the priority level of a thread. This value is a signed delta from the base priority level assigned to the thread.

dwFlags

Reserved; do not use.

VWIN32_DIOCCompletionRoutine Group

Group

Notifies the system that an asynchronous operation in a virtual device driver is complete. The VxD calls this service after the asynchronous input and output (I/O) operation to signal the application.

```
mov ebx, Internal  ; see below
VxDCall VWIN32_DIOCCompletionRoutine
```

Parameters

Internal

Event identifier. This parameter must be the same value initially passed to the VxD in the **Internal** member of the **OVERLAPPED** structure.

Return Value

No return value.

Remarks

Before calling this service, the VxD must set the **InternalHigh** member of the **OVERLAPPED** structure to the number of bytes of return data.

W32_DEVICEIOCONTROL Group

Group

Passes a control code and related information to a virtual device driver.

```
include VMM.INC

mov ebx, VMHandle
mov eax, W32_DEVICEIOCONTROL
mov esi, OFFSET32 dioparams
VMMCall System_Control
```

Parameters

VMHandle

Handle of the virtual machine.

dioparams

Address of a **DIOCParams** structure containing a control code and information that the VxD needs to process the control code.

Return Value

Returns one of the following values in the EAX register:

0	The control code processed successfully.
- 1	An asynchronous operation is in progress. A VxD must return this value only if the **lpoOverlapped** member of the **DIOCParams** structure is not NULL.
Error code	An error occurred.

Remarks

This message is sent to a VxD when an application specifies the name of a VxD in the **CreateFile** function and when an application specifies the device handle of the VxD in a call to the **DeviceIoControl** or **CloseHandle** function. This message uses the ESI register.