Introduction

Programming Techniques contains topics that are of interest to programmers writing Win32-based applications. It is a supplement to the Win32 Programmer's Reference and the documentation included with your development tools. It is not a comprehensive guide.

Programming techniques contains the following sections:

Win32 Application Programming Interface

This section provides an overview of the Microsoft Win32 Application Programming Interface (API), how it is organized, and some of the features that you can use in your Win32-based applications.

About the Win32 API

The Win32 API allows applications to exploit the power of the 32 bit Windows family of operating systems. The functions, structures, messages, macros, and interfaces form a consistent and uniform API for all of Microsoft's 32-bit platforms. Using the Win32 API, you can develop applications that run successfully on all platforms while still being able to take advantage of unique features and capabilities of any given platform.

Differences in the implementation of the programming elements depend on the capabilities of the underlying features of the platform. The most notable difference is that some functions carry out their tasks only on the more powerful platforms. For example, security functions are only available on the Windows NT operating system. Most other differences are system limitations, such as restrictions on the range of values or the number of items a given function can manage. For more information about system limitations in Windows 95, see Windows 95 System Limitations. For more information about system limitations in Win32s, see the Win32s Programmer's Reference.

The Win32 API can be grouped into these functional categories:

- Window Management
- Window Controls
- Shell Features
- Graphics Device Interface
- System Services
- International Features
- Network Services

Window Management

The window management functions give applications the means to create and manage a user interface. You use the window management functions to create and use windows to display output, prompt for user input, and carry out the other tasks necessary to support interaction with the user. Most applications create at least one window.
Applications define the general behavior and appearance of their windows by creating window classes and corresponding window procedures. The window class identifies default characteristics, such as whether the window processes double clicks of the mouse buttons or has a menu. The window procedure contains the code that defines the behavior of the window, carries out requested tasks, and processes user input.
Applications generate output for a window using the GDI functions. Because all windows share the display screen, applications do not receive access to the entire screen. Instead, the system manages all output so that it is aligned and clipped to fit within the corresponding window. Applications can draw in a window in response to a request from the system or while processing input messages. When the size or position of a window changes, the system typically sends a message to the application requesting that it paint any previously unexposed area of its window.
Applications receive mouse and keyboard input in the form of messages. The system translates mouse movement, mouse button clicks, and keystrokes into input messages and places these messages in the message queue for the application. The system automatically provides a queue for each application. The application uses message functions to extract messages from the queue and dispatch them to the appropriate window procedure for processing.
Applications can process the mouse and keyboard input directly or let the system translate this low-level input into command messages by using menus and keyboard accelerators. You use menus to present a list of commands to the user. The system manages all the actions required to let the user choose a command and then sends a message identifying the choice to the window procedure. Keyboard accelerators are application-defined combinations of keystrokes that the system translates into messages. Accelerators typically correspond to commands in a menu and generate the same messages.
Applications often respond to command messages by prompting the user for additional information with dialog boxes. A dialog box is a temporary window that displays information or requests input. A dialog box typically includes controls − small, single-purpose windows − that represent buttons and boxes through which the user makes choices or enters information. There are controls for entering text, scrolling text, selecting items from a list of items, and so on. Dialog boxes manage and process the input from these controls, making this information available to the application so that it can complete the requested command.
You can share useful data, such as bitmaps, icons, fonts, and strings, by adding this data as "resources" to the file for an application or DLL. Applications retrieve the data by using the resource functions to locate the resources and load them into memory.
Window management functions provide other features related to windows, such as carets, the clipboard, cursors, hooks, icons, and menus.

Window Controls

The shell incorporates a number of controls that help give Windows its distinctive look and feel. Because these controls are supported by DLLs that are a part of the operating system, they are available to all applications. Using the common controls helps keep an application's user interface consistent with that of the shell and other applications. Because developing a control can be a substantial undertaking, using the common controls can also save you a significant amount of development time.

The common controls are a set of control windows that are supported by the common control library, COMCTL32.DLL. Like other controls, a common control is a child window that an application uses in conjunction with another window to perform I/O tasks. The common control DLL includes a programming interface that applications use to create and manipulate the controls as well as to receive user input from them.

Shell Features

The Win32 API contains a number of interfaces and functions that applications can use to enhance various aspects of the shell.

A namespace is a collection of symbols, such as file and directory names or database keys. The shell uses a single hierarchical namespace to organize all objects of interest to the user, including files, storage devices, printers, and network resources. The namespace is similar to the directory structure of a file system, except that the namespace contains objects other than files and directories.

A shortcut (also called a shell link) is a data object that contains information used to access another object located anywhere in the shell's namespace. A shortcut allows an application to access an object without having to know the current name and location of the object. Objects that can be accessed through shortcuts include files, folders, disk drives, printers, and network resources.

There are several ways to extend the shell. The system uses icons to represent files in the shell's namespace. By default, the system displays the same icon for all files that have the same filename extension. An icon handler can override the default and set the icon for a particular file. A context menu handler is a shell extension that modifies the contents of a context menu. The system displays a context menu when the user clicks or drags an object using mouse button 2. The context menu contains commands that apply specifically to the object that was clicked or dragged. Most context menus have a Properties command that displays the property sheet for the selected item. A property sheet contains information about an object in a set of overlapping windows called pages. A property sheet handler is a shell extension that adds pages to a system-defined property sheet or replaces pages in a Control Panel application's property sheet. A copy hook handler is a shell extension that approves or disapproves the moving, copying, deleting, or renaming of a file object.

The shell includes a Quick View command that allows the user to view the contents of a file without having to run the application that created it. A file viewer provides a user interface for viewing a file. The shell uses the filename extension to determine which viewer to run. You can provide file viewers for new file formats or replace an existing viewer with one that includes more functionality. A file viewer works in conjunction with a file parser, which provides the parsing needed to generate the Quick View of a file of a given type. You can provide additional file parsers to support new file types.

Graphics Device Interface

The graphics device interface (GDI) provides functions and related structures that an application can use to generate graphical output for displays, printers, and other devices. Using GDI functions, you can draw lines, curves, closed figures, paths, text, and bitmap images. The color and style of the items you draw depends on the drawing objects − that is, pens, brushes, and fonts − that you create. You can use pens to draw lines and curves, brushes to fill the interiors of closed figures, and fonts to write text.

Applications direct output to a given device by creating a device context (DC) for the device. The device context is a GDI-managed structure containing information about the device, such as its operating modes and current selections. An application creates a DC by using device context functions. GDI returns a device context handle, which is used in subsequent calls to identify the device. For example, using the handle, an application can retrieve information about the capabilities of the device, such as its technology type (display, printer, or other device) and the dimensions and resolution of the display surface.

Applications can direct output to a physical device, such as a display or printer, or to a "logical" device, such as a memory device or metafile. Logical devices give applications the means to store output in a form that is easy to send subsequently to a physical device. Once an application records output in a metafile, it can play that metafile any number of times, sending the output to any number of physical devices.

Applications use attribute functions to set the operating modes and current selections for the device. The operating modes include the text and background colors, the mixing mode (also called the binary raster operation) that specifies how colors in a pen or brush combine with colors already on the display surface, and the mapping mode that specifies how GDI maps the coordinates used by the application to the coordinate system of the device. The current selections identify which drawing objects are used when drawing output.

System Services

System services functions that give applications access to the resources of the computer and the features of the underlying operating system, such as memory, file systems, devices, processes, and threads. An application uses system services functions to manage and monitor the resources that it needs to complete its work. For example, an application uses memory management functions to allocate and free memory and uses process management and synchronization functions to start and coordinate the operation of multiple applications or multiple threads of execution within a single application.

System services functions provide access to files, directories, and input and output (I/O) devices. The file I/O functions give applications access to files and directories on disks and other the storage devices on a given computer and on computers in a network. These functions support a variety of file systems, from the FAT file system to the CD-ROM file system (CDFS) to the New Technology File System (NTFS).

System services functions provide methods for applications to share code or information with other applications. For example, you can make useful procedures available to all applications by placing these procedures in DLLs. Applications access these procedures by using DLL functions to load the libraries and retrieve the addresses of the procedures. Communications functions read from and write to communications ports as well as control the operating modes of these ports. There are several methods of interprocess communication (IPC), such as DDE, pipes, mailslots, and file mapping. For operating systems that provide security features, the security functions give applications access to secure data as well as protect data from intentional or unintentional access or damage.

System services functions provide access to information about the system and other applications. System information functions let applications determine specific characteristic about the computer, such as whether a mouse is present and what dimensions elements of the screen have. Registry and initialization functions let applications store application-specific information in system files so that new instances of the application or even other applications can retrieve and use the information.

System services functions provide features that applications can use to handle special conditions during execution, such as handling errors, logging events, and handling exceptions. There are features that applications can use to debug and improve performance. For example, debugging functions permit single-step control of the execution of other processes, and performance monitoring allows for detailing the path of execution through a process.

System services functions provice features you can use to create other types of applications, such as console applications and services.

International Features

These features are available to help you write applications for international markets. The Unicode character set uses 16-bit character values to represent characters used in computing, such as technical symbols, and many written languages. The national languages support (NLS) functions help you localize your application for international markets. The input method editor (IME) functions, available in Asian versions of Windows, help users enter text containing Unicode and DCBS characters.

Network Services

The network functions allow communication between applications on different computers on a network.
The network functions create and manage connections to shared resources, such as directories and network printers, on computers in the network.
The network interfaces include Windows Networking, Windows Sockets, NetBIOS, RAS, SNMP, and Network DDE.
Windows NT also supports many of the LAN Manager functions. Windows 95 supports a small subset of these functions.

Win32 Platform Differences

The following table contains a variety of Win32 API and operating system features and how each platform supports them. Bold text indicates an operating system feature, rather than an API feature.

Feature	Windows NT	Windows 95	Windows 3.1 with Win32s
32-bit Coordinate System	Yes	No	No
32-bit Flat Memory model	Yes	Yes	Yes
3-D look	Yes	Yes	Yes
Asynchronous file I/O	Yes	No	No
Asynchronous input model	Yes	Yes	No
COMM	Yes	Yes	Via Universal Thunk
Common Controls			
Property sheet tabs	Yes	Yes	Yes
Drag list boxes	Yes	Yes	Yes
Toolbar	Yes	Yes	Yes
Status bar	Yes	Yes	Yes
Column heading	Yes	Yes	Yes
Spin buttons	Yes	Yes	Yes
Slider	Yes	Yes	Yes
Scrolling button indicator	Yes	Yes	Yes
Rich Edit Control	Yes	Yes	Yes
Progress indicator	Yes	Yes	Yes
Tree View	Yes	Yes	Yes
List View	Yes	Yes	Yes
Common Dialogs	Yes	Yes	Yes
Console Support	Yes	Except code page	No
Context menu on mouse button 2	Yes	Yes	Yes
Enhanced metafiles	Yes	Yes	No
Event logging	Yes	No	No
File mapping	Yes	Yes	Yes
File merge/reconciliation	Not yet	Yes	No
File Viewers	Yes	Yes	Not used
Image Color Matching (ICM)	Not yet	Yes	No

Long Filenames (LFN)	Yes	Yes	Won't appear
Multimedia API	Yes	Yes	Windows 3.1 level
Multiprocessor Machines	Yes	No	No
Named pipes	Yes	Client-side	Client-side
National Language Support (NLS)	Yes	Yes	Yes
Network DDE	Yes	Via Thunk	No
Non-Intel machines	Yes	No	No
Paths/Beziers	Yes	Limited	No
Pen	No	Yes	No
Plug and Play event aware	Won't get events	Yes	Won't get events
Preemptive multitasking	Yes	Yes	No
Print spooler	Yes	Except forms	No
Remote Access Services (RAS)	Yes	Yes	No
Remote Procedure Calls (RPC)	Yes	Yes	Via thunk
Security	Yes	No	No
Security (C2 certifiable)	Yes	No	No
Separate address space	Yes	Yes	No
Service control manager	Yes	No	No
Simple MAPI	Yes	Yes	Via Universal Thunk
Structured exception handling (SEH)	Yes	Yes	Yes
TAPI	Yes	Yes	No
Threads	Yes	Yes	No
Unicode	Yes	No	No
Universal Naming Convention (UNC)	Yes	Yes	Yes
User & GDI system resources	Virtually Unlimited	Expanded	Win3.1 limits
Windows 4.0 help	Yes	Yes	Context menu help won't appear
Windows Network (WNet)	Yes	Yes	Via Universal Thunk
Windows Sockets	Yes	Yes	Yes
World transforms	Yes	Scaling Only	No

Writing Great Win32-based Applications

The Microsoft® Windows® user interface is based on a datacentric design; that is, rather than focusing on applications, the user interface design emphasizes data and tasks that involve the manipulation of data. The interface has been designed to allow the user to browse for data and documents and to edit them directly without necessarily having to locate an appropriate editor or application first. This type of design frees the user to focus on information and tasks rather than on applications and how they interact.

About Writing Great Win32-based Applications

This overview briefly describes some of the features you should use and guidelines you should follow to ensure that your application is a "great" Win32 - based application. A great Win32 - based application is one that integrates seamlessly with the user interface in Windows and conforms to the system's datacentric design principles. In addition to reading this article, you should follow the user interface guidelines presented in *The Windows Interface Guidelines for Software Design*.

You should use the following features:

- File Information
- Long Filenames
- Context Menus
- Icons
- Shortcuts
- Clipboard Data Transfer Operations
- Common Controls and Dialog Boxes
- Palette
- Windows Help
- Multiple Views
- Pen Input

File Information

Throughout the shell, files appear as icons. When you click on a file's icon using mouse button 2, the system displays a context menu containing commands that perform actions on the file. One of the commands, Properties, displays a special dialog box called a property sheet that contains information about the file. By viewing a file's property sheet, the user can find out information about a file without having to open it.

By default, a file's property sheet contains general information about the file, including its name, size, location, creation date, attributes, and so on. The following illustration shows the default property sheet for a typical file.

{ewc msdncd, EWGraphic, bsd23546 0 /a "SDK5GUIDE_01.BMP"}

If your application creates files with additional properties that the user may be interested in, you should add more pages to the property sheets for the files. One way to add property sheet pages to an application using OLE structured storage is to store documents in compound files (also called docfiles) and use the Document Summary Information Property Set to store summary information and editing statistics for the documents. When the user activates the property sheet for the document, the shell automatically gathers the summary information and editing statistics from the document and adds them to the property sheet as two additional pages. The following illustration shows a property sheet with Summary and Statistics pages added based on data gathered from the document.

{ewc msdncd, EWGraphic, bsd23546 1 /a "SDK5GUIDE_02.BMP"}

For more information about saving document information using the OLE Document Summary Information Property Set, see the *OLE Programmer's Reference*.

Another way to add pages to file property sheets is to write a shell extension (OLE InProcServer32) that includes a property sheet handler. Whenever the user activates the property sheet for a file, the system checks the registry to see if any property sheet handlers are registered for the file type. If there are some registered, the system calls the handlers before displaying the property sheet. The handlers can add any number of pages to the property sheet before it is displayed. For more information about shell extensions and property sheet handlers, see Shell Extensions.

Long Filenames

Windows allows users and applications to create and use long filenames for their files and directories. A long filename is a name for a file or directory that is longer than the standard MS-DOS filename format (8 character base, 3 character extension).

An application should support long filenames and display them correctly. You can use the **SHGetFileInfo** function in your application to retrieve the long filename for a file as well as the file's icon, type name, attributes, and so on.

If you include the File Open and Save As common dialog boxes in your application, you can use the OFN_LONGNAMES value to direct the dialog boxes to display and return long filenames. Before an application displays a long filename, it should hide the filename extension. For example, the application should display a filename like "My letter to Mom" instead of "My letter to Mom.Doc." An application can hide filename extensions on a file-specific basis by using the **SHGetFileInfo** function.

The following illustration shows a folder containing documents with long filenames.

{ewc msdncd, EWGraphic, bsd23546 2 /a "SDK5GUIDE_03.BMP"}

If an application is used to view or edit a document or data file, the title bar of the window that contains the file should display the long filename for the file. If the title bar also includes the application's name, it should appear to the right of the filename. Displaying the filename first places the emphasis on the document or data rather than on the application. For more information about long filenames, see Long Filenames.

You should also support Universal Naming Convention (UNC) path names for files in your application. Using UNC names enables users to browse documents on the network directly and to open an application's files on remote machines without needing to know the location of the file on the network or having to make an explicit network connection.

Context Menus

A context menu is a pop-up menu containing a set of commands that are specific to a particular object. Windows provides a context menu for all objects that appear in the shell, including files, folders, printers, and so on. A context menu appears when the user clicks an object using mouse button 2. Because context menus are displayed at the pointer's current location, they eliminate the need for the user to move the pointer to the menu bar or toolbar. They also help eliminate screen clutter.

You should provide context menus for all objects in an application and should display the context menu whenever the user clicks an object using mouse button 2. Each context menu should include a Properties command that displays a property sheet for the object.

In addition to displaying a context menu for objects, an application should also display a context menu when the user clicks the small icon in the title bar using mouse button 2. The commands in the context menu should operate on the object that is open in the window, not on the window itself. To see an example of a context menu associated with a title bar icon, click the title bar icon of a folder window in the shell using mouse button 2. For more information about context menus, see Shell Extensions.

Icons

If your application supports OLE, you should make sure that the icons for your embedded and linked objects are consistent with the shell. For example, when the user drags an icon from the shell into your container, the icon and its name should stay the same.
You should support interactions with embedded icons the same way that the shell does. For example, when the user selects the icon for an embedded object, you should dither the icon with the system highlight color rather than enclosing it in a rectangle that has resizing handles.

Shortcuts

A shortcut (also called a shell link) is a data object that contains information used to access another object in the system, such as a file, folder, disk drive, or printer. A shortcut has an icon associated with it; the user accesses the object associated with a shortcut by double-clicking the shortcut's icon. The associated object can be stored anywhere in the system.

Typically, the user creates shortcuts to gain quick access to objects stored in subfolders on the same machine or to shared folders on other machines. For example, the user can create a shortcut to a Microsoft Word document located in a subfolder and can place the shortcut icon on the desktop. The user can later start Word and open the document simply by double-clicking the shortcut icon. If the document is later moved or renamed, the system takes steps to update the shortcut the next time the user selects it.

An application should support shortcuts. For example, a word processing application might allow the user to drag and drop a shortcut icon into a document file. An application should also correctly dereference shortcuts. For example, if the user specifies the filename of a shortcut when using an application's Open command on the File menu, the application should open the object associated with the shortcut, not the shortcut file itself. For more information about shortcuts, see Shell Links.

Clipboard Data Transfer Operations

Windows supports two types of clipboard data transfer operations − those involving menu commands (such as Cut, Copy, and Paste), and those involving the direct manipulation of objects (drag and drop). An application should support both types extensively.

You should support the OLE style of drag and drop. If you support drag and drop, the user can easily move data among the desktop, folders, and other applications. You should support dragging with mouse button 2 and display a context menu at the end of the drag operation, as the shell does. At a minimum, the menu should include these commands: Move Here, Copy Here, Create Shortcut(s) Here, and Cancel. For more information about supporting the OLE style of drag and drop, see Dragging and Dropping.

You should make sure your application's menu-based data transfer model works well with the shell. You should test various scenarios, such as copying a shortcut or file in a shell folder to the clipboard and then pasting the shortcut or file into your application. Also, if your application supports shortcuts to its documents, you should offer a link to your OLE data object when the user drags an object out of a document.

Common Controls and Dialog Boxes

The shell incorporates a number of control windows and dialog boxes that help give Windows its distinctive look and feel. Because these controls and dialog boxes are supported by DLLs that are a part of Windows, they are available to all applications. You should use the common controls and dialog boxes − rather than developing similar controls and dialog boxes of your own − because they help keep your application's user interface consistent with that of the shell and other applications. Because developing a control or dialog box can be a substantial undertaking, using the common controls and dialog boxes can also save you a significant amount of development time.

Common Controls

The common controls are a set of control windows that are supported by the common control library, which is a DLL called COMCTRL32.DLL. Like other control windows, a common control is a child window that an application uses in conjunction with another window to perform input and output (I/O) tasks. The common control DLL includes a programming interface that you use to create and manipulate the controls and dialog boxes and to receive user input from them. This section describes some of the controls provided by the common control DLL.

Header Control

A header control provides headings for columns of text or numbers. It can be divided into many parts, and each part can have its own heading text. The user can adjust the width of the columns by dragging the dividers that separate the parts.
{ewc msdncd, EWGraphic, bsd23546 3 /a "SDK5GUIDE_05.BMP"}

Property Sheet

A property sheet displays the properties of an object, such as a document file or a cell in a spreadsheet. Related properties can be grouped together and placed on separate, overlapping pages within the property sheet. Each page has a tab that the user can select to bring the page to the foreground.

An application can create a special type of property sheet called a wizard control. The control displays a sequence of pages that guide the user through the steps of an operation, such as setting up a device or creating a birthday card.
{ewc msdncd, EWGraphic, bsd23546 4 /a "SDK5GUIDE_08.BMP"}

Up-Down Control

An up-down control consists of a pair of arrow buttons that the user can click to increment or decrement a value, such as a scroll position or a number displayed in an accompanying edit control.
{ewc msdncd, EWGraphic, bsd23546 5 /a "SDK5GUIDE_13.BMP"}

Tree View Control

A tree view control displays a hierarchical list of items, such as the headings in a document, the entries in an index, or the files and directories on a disk. By clicking an item, the user can expand or collapse the associated list of subordinate items. The user can select items, edit item labels, and drag items from one location to another.
{ewc msdncd, EWGraphic, bsd23546 6 /a "SDK5GUIDE_12.BMP"}

Status Window

A status window displays information that may be useful to the user. It is typically positioned along the bottom of a window and can be divided into parts to display different types of information simultaneously.
{ewc msdncd, EWGraphic, bsd23546 7 /a "SDK5GUIDE_09.BMP"}

Toolbar

A toolbar contains buttons that carry out commands when the user selects them. Typically, the buttons correspond to menu items, providing a quicker, more direct way for the user to access an application's commands.
{ewc msdncd, EWGraphic, bsd23546 8 /a "SDK5GUIDE_10.BMP"}

Progress Bar

A progress bar indicates the progress of a lengthy operation. It consists of a rectangle that is gradually filled with color, from left to right, as the operation progresses.
{ewc msdncd, EWGraphic, bsd23546 9 /a "SDK5GUIDE_07.BMP"}

List View Control

A list view control displays a collection of related items, each consisting of an icon and a descriptive label. The items can be arranged and displayed in different ways to suit the user's preferences. The user can select items, edit item labels, and drag items from one location to another.
{ewc msdncd, EWGraphic, bsd23546 10 /a "SDK5GUIDE_06.BMP"}

Trackbar

A trackbar allows the user to select a value from a range of consecutive values. To select a value, the user drags the trackbar's slider to the desired position.
{ewc msdncd, EWGraphic, bsd23546 11 /a "SDK5GUIDE_11.BMP"}

Common Dialog Boxes

Windows provides several common dialog boxes that your application can use to obtain various kinds of information from the user. There are the following types of common dialog boxes.

Dialog box	Description
Color	Enables the user to select and create colors.
Find	Enables the user to specify a search string.
Open	Enables the user to specify the location and filename of a file to be opened.
Page Setup	Enables the user to set the attributes of a printed page, including the paper type, the paper source, the page orientation, and the width of the page margins.
Print	Enables the user to configure a printer for a particular print job. The user can set print job parameters, such as the print quality, print range, and number of copies.
Replace	Enables the user to specify strings for use in a search and replace operation.
Save As	Enables the user to specify the location and name of a file to be saved.

Note that the Print Setup dialog box provided in previous versions of Windows is now obsolete; new applications should use the Page Setup dialog box.

The Open and Save As common dialog boxes accessed from the File menu are especially useful because they support many features of the shell, including shell links, long filenames, and direct browsing of the network. If you cannot use the Open and Save As dialog boxes, you should incorporate the following features into your open and save dialog boxes to ensure that they are consistent with the shell, the Windows accessories, and other applications:

- Support the same namespace hierarchy as the shell; that is, Desktop should be at the root of the hierarchy, followed by all folders and objects on the desktop, including My Computer, My Network, and so on. For more information about the shell namespace, see Shell's Namespace.
- Support shortcuts (also known as shell links). Note that opening a shortcut should open the target of the shortcut rather than the shortcut file itself. For more information about shortcuts, see Shell Links.
- Display filenames with the corresponding icons and filename extensions removed, as the shell does.
- Allow the user to browse the network hierarchy directly.
- Make sure that all of your dialog boxes (not just your open and save dialog boxes) use only nonbold fonts. In addition, you should use the DS_3DLOOK style to give your dialog boxes the three-dimensional look used throughout the system.

Palette

If your application supports 256 colors, you should use the Windows halftone palette, as the shell does. It helps system performance, because the system does not need to load a new palette every time the execution focus switches between the shell and your application.

Windows Help

Windows Help includes many features that you can use to provide help information that is task- or object-specific as well as readily accessible and unobtrusive. You should consider including the following features in your application's help file:

- Provide context-sensitive help for your dialog boxes and documents. The user can access context-sensitive help by clicking mouse button 2 (if there is no context menu available) or pressing the F1 key to display help information for a specific object or element in the application. The following illustration shows context-sensitive help for a control in a dialog box.

{ewc msdncd, EWGraphic, bsd23546 12 /a "SDK5GUIDE_04.BMP"}

- Use secondary windows for procedural help. A secondary window does not have menus, and it remains open until it is explicitly closed.
- Embed shortcut buttons in your help text. A shortcut button allows the user to start an application from within a help file.
- Use sizable topic windows to make help text easier to read.
- Consider using the built-in support for training cards.

Multiple Views

You should not let the user open multiple views of the same document. Multiple views confuse the user and conflict with the datacentric design of Windows.
When the user attempts to open a document file associated with an application, typically by double-clicking the document file's icon, the application should determine if the document file is already open. If it is, the application should check whether the current user has attempted to open the file. The application should do more than just compare user names because the user may be logged onto more than one computer. If the current user already has the document file open, the application should immediately restore the window containing the open document file. If the current user has not attempted to open the document file (meaning that someone else on the network has), the application should prompt the user with the following message.

```
This document has already been opened by <name>. Would you like to make a
copy?
```

If the user does not want to make a copy, the application should exit; otherwise, it should make a copy.
You should also handle the case where the user double-clicks an application's icon when the application is already running. If the application's default action is to open a blank document when the user double-clicks the icon, the application should present the user with a list of currently opened documents. Opening a new document, however, should be the default action.

Pen Input

Windows 95 only
An application should support pen input so that it is easy to use on pen-based platforms, such as notebook computers and desktop tablets. The following list briefly describes what you need to do to support pen input:

- Use functions from the Windows pen application programming interface (API) to activate the pen in your application. If you activate the pen, the user can enter text using handwriting recognition and edit documents using gestures.
- Incorporate ink-edit controls into your application. Ink-edit controls allow the user to enter scribbled notes, drawings, and signatures.
- Add other natural pen-oriented features and gestures to your application.

Window 95 Logo Requirements

This section describes the technical requirements that software programs and hardware devices must meet to qualify for the Windows 95 Logo. These requirements are periodically updated.

General Requirements for Applications

The requirements for the Windows 95 Logo apply to the following four main types of programs:

- File-based applications − that is, applications that provide Open, Save, and Close File menu options.
- Applications that are not file-based and applications that run exclusively in full screen mode. An application that runs exclusively in full screen mode is one that cannot run in a window or be minimized.
- Utilities (for example, virus scanners and disk management programs).
- Development tools (for example, compilers and linkers).

To qualify for the Windows 95 Logo, an application must meet the appropriate requirements in the following list. The first five requirements apply to all types of applications.

1. An application must use the Win32 application programming interface (API) and must be compiled using a 32-bit compiler that generates an executable file of the Portable Executable (PE) format. If your application is not represented in the PE format (for example, it uses interpreted code), the "run-time engine" must be an executable file in the PE format. For example, if you develop an application in Microsoft Access, your application is an .MDB file, not an .EXE, but ACCESS.EXE would need to be a PE format executable file.
2. An application must support the new shell and user interface. At a minimum, an application must meet the following requirements:
   - Register both 16- by 16-pixel and 32- by 32-pixel icons for each file type and the application.
   - Follow the user interface guidelines described in *The Windows Interface Guidelines for Software Design*. An application should also use the system-defined dialog boxes and controls.
   - Use the system metrics for setting the size of elements within the application.
   - Use the system-defined colors.
   - Use the right mouse button for context menus (and *not* for any other purpose).
   - Follow the application installation guidelines to make the application properly visible in the shell. At a minimum, this means that you should use the registry, you should not add information to the WIN.INI or SYSTEM.INI file, and you should provide complete uninstall capabilities with your application. In addition, the installation process must be automated. For more information about installation guidelines, see Installing Applications.

   For detailed guidelines about supporting the shell and user interface, see *The Windows Interface Guidelines for Software Design*.
3. An application must be tested on the latest version of Microsoft® Windows NT®. If it uses features available only in one platform, the application must handle this gracefully when running on the other platofmr.
4. An application must support long filenames and use them to display all document and data filenames in the shell, in title bars, in dialog boxes and controls, and with icons. In addition, an application should hide the extensions of filenames that are displayed within the application itself.
5. An application should process Plug and Play events. For example, it should be aware of slow links and should react to system messages that occur when a new device is attached or removed.

The next three requirements apply to file-based applications that do not run in full screen mode. Some games and children's software run exclusively in full screen mode and *need not follow* these three requirements.

6. An application must support Universal Naming Conventions (UNC) names for paths.
7. An application must support OLE containers or objects, or both. It must also support the OLE style of drag and drop. An application should also support OLE automation and compound files (with document summary information included).
8. An application must support simple mail enabling using the Messaging Application Programming Interface (MAPI) or the Common Messaging Call (CMC) API. That is, it must include Send Mail functionality.

The following items are modified requirements for utilities, such as disk optimizers and anti-virus software:

9. Same as number 1, except for components that must use exclusive volume locking functions, soft interrupts, or components that must talk directly to 16-bit drivers. The user interface and other components of these

applications must be 32 bits and use the Windows 95 thunking mechanism to access the 16-bit components.

10. Same as number 2.
11. Same as number 3, except for products like disk utilities that implement platform-specific functionality that does not make sense in Windows NT.
12. Same as number 4.
13. Numbers 5 through 8 are recommended, but not required. However, number 6 *is* required if your product accesses network resources.

The following items are modified requirements for compilers and other development tools:

14. In addition to the requirements that follow, if Windows is one of the target platforms of a compiler or development tool, the compiler or tool must be capable of generating applications that can meet all of the Windows 95 Logo requirements.
15. Same as number 1.
16. Same as number 2, except that when icons are registered for each file type and the application, common source filename extensions, such as .C, .CPP, .H, and .HPP, are excluded.
17. Same as number 3.
18. Same as number 4.
19. Same as number 5.
20. Same as number 6.
21. Compilers and development tools must support OLE in the following ways:
    - Support the OLE style of drag and drop (recommended within the tool's design environment).
    - Support OLE automation (recommended, but not required).
    - Provide an easy way to create applications with OLE container or object support, or provide this functionality by default.
22. Same as number 8 (recommended, but not required).

Personal Computer Systems

For a personal computer (PC) system to qualify for the Windows 95 Logo, it must meet a minimum set of requirements as outlined below and pass the System Compatibility Test (SCT). The SCT tests are included in the Microsoft Windows 95 Device Driver Kit (DDK), along with instructions for OEM participation. System testing is OEM-administered, and results are sent to Microsoft Compatibility Labs (MCL).
A PC system must meet the following requirements:

1. 80386 or compatible processor. (However, 33-megahertz 80486 or better is recommended.)
2. 4 megabytes (MB) random-access memory (RAM). (However, 8 MB is recommended.)
3. Plug and Play basic input/output system (BIOS) version 1.0a or later that reads back all resources. (However, a BIOS that soft-sets all resources is recommended.)
4. Molded-in or permanently printed icon labels on the computer case for built-in ports. Ideally, icons on the cable connectors should match the icons on the computer case.
5. Optional read-only memory (ROM) chips on expansion cards must use the Plug and Play header format documented in the Plug and Play BIOS specification.
6. A Video Graphics Array (VGA) display adapter that uses a packed-pixel frame buffer and provides a resolution of at least 640 by 480 pixels and 8 bits per pixel (bpp) for desktop systems and a 64-shade gray scale for mobile systems. (However, VGA 1024 by 768 pixels and 8 bpp is recommended for desktop systems, and 64 colors is recommended for mobile systems.)
7. One parallel port that supports IEEE-P1284-I mode protocols for compatibility mode and nibble mode. The system must be capable of receiving the parallel device's identifier in nibble mode. (However, ECP P1284-I is recommended.)
8. One integrated or separate serial port, with 1-16550A required for mobile systems. Also recommended are 1-16550A for desktop systems, an additional PS/2 style port, pen devices with a barrel button, and serial infrared devices meeting the Infrared Data Association (IrDA) specification.
9. Advanced Power Management (APM) version 1.1 is required for mobile systems. (However, it is recommended also for desktop systems.)

If the system ships with expansion cards or peripheral devices integrated onto the motherboard, it is recommended that the cards or devices meet the Windows 95 Logo specifications defined in this article and use 32-bit Windows 95 - based device drivers.
For more information about qualifying a PC for the Windows 95 Logo, see the *Hardware Design Guide for Windows 95*.

Hardware Peripheral Devices

For a peripheral device to qualify for the Windows 95 Logo, it must meet the requirements described in the *Hardware Design Guide for Windows 95* and pass the compatibility tests conducted by MCL. For information about prequalifying test tools and MCL device and driver submission details, see the Windows 95 DDK. The Windows 95 DDK also contains detailed information about designing Windows 95 - based device drivers.
To carry the Windows 95 Logo, a device driver must support the following Plug and Play capabilities in Windows 95:

1. Retrieves configuration information from Configuration Manager.
2. Is dynamically loadable.
3. Is dynamically reconfigurable.
4. Reacts to system messages that occur when a device is attached or removed.

An ideal Windows 95 - based Plug and Play driver requires minimal user interaction to select the proper driver. In addition, the settings for the device may need to change based on which user is logged in, whether the machine is docked or not, or both.

Display Adapters

Display adapters must meet the following requirements:

1. Support the VGA graphics standard.
2. Support at least a 640- by 480-pixel, 8 bpp display driver. Desktop systems must be able to display at least 256 colors, and mobile systems must support an 8 bpp driver and map colors into at least a 64 gray scale display so that changes to higher-resolution external monitors can be made without restarting Windows 95.
3. Use a packed-pixel frame buffer with at least 8 bpp.
4. Use a VGA BIOS that, if it exists separately, has its base address fixed at C000h. (However, an alternate address is recommended.)
5. Use a standard VGA with a page frame and I/O address resource that can be static − that is, not relocatable.
6. Support the Video Electronics Standards Association (VESA) ergonomic timings.
7. Be capable of being disabled if a conflicting VGA expansion card is added to the system.
8. Provide at least one alternate configuration in case of conflict during initial program load (IPL) boot (non-VGA display resources only). The VGA BIOS must be able to use alternate configuration register addresses.
9. Have the display adapter circuitry come up active when power is turned on or the system is reset. This requirement applies only to an Industry Standard Architecture (ISA) Plug and Play display adapter expansion cards used as a system boot device.

Audio Adapters

Audio adapters must meet the following requirements:

1. Be able to produce 22 kilohertz (kHz), 8-bit, monaural, output-only sound (minimum performance).
2. Support either Sound Blaster or the Microsoft Windows Sound System to use built-in drivers for Windows 95.
3. Use a one-eighth inch miniature phone jack wired for stereo as the output connector.
4. Map the base input and output (I/O) address to configurations compatible with either Sound Blaster or the Microsoft Sound System.
5. Support at least all interrupt request (IRQ) signals used either by Sound Blaster or the Microsoft Windows Sound System.
6. Support the selection of at least three available Direct Memory Access (DMA) channels, either 8 bit or 16 bit, if DMA is supported.
7. Support disabling in case of resource conflicts with other devices.

Storage Devices

This section lists the requirements for storage devices, including floppy disk controllers, ATA (IDE) adapters, ATA (IDE) peripherals, small computer system interface (SCSI) host adapters, and SCSI devices.

Floppy Disk Controllers

Floppy disk controllers must meet the following requirements:

1. Use at least three static I/O addresses: 3F2h, 3F4h, and 3F5h.
2. Support IRQ6.
3. Support at least DMA 2, if DMA is used. In addition, the controller should be capable of selecting at least two other available DMA channels, either 8 bit or 16 bit.
4. Be capable of being independently disabled.

ATA (IDE) Adapters

ATA (IDE) adapters must meet the following requirements:

1. Use the first device attached to the adapter as the boot device.
2. Use the standard I/O addresses: 1F0h through 1F7h and 3F6h.
3. Support at least IRQ14.
4. Be capable of being disabled if an ATA (IDE) expansion card is added to the system. In addition, if a single adapter card contains a floppy disk drive controller, the adapter must be able to independently disable the floppy drive controller if a conflict occurs.

ATA (IDE) Peripherals

ATA (IDE) peripherals must meet the following requirements:

1. Support the ATA Packet Interface (ATAPI) protocol for CD-ROMs defined in SFF-8020 version 1.2.
2. Comply with the requirements specified in the ATA 2 specification.
3. Set the signature after an ATA Read or ATA Identify Command is received.
4. Implement the SEEK command and set the DSC bit when the ATAPI seek is complete, but not change the drive select bit.
5. Return the CANNOT READ MEDIUM - INCOMPATIBLE FORMAT additional sense code qualifier when a READ is received on an audio track.
6. Support CD-DA.
7. Support the READ_CD command sector types mode 2 form 1, mode 2 form 2, mode 1 form 1, and mode 1 form 2.
8. Support the Test_Unit_Ready command.

SCSI Host Adapters

SCSI host adapters must meet the following requirements:

1. Meet the standards described in the current version of the Plug and Play SCSI specification.
2. Support the SCSI Configured Auto-Magically (SCAM) Level 1 protocol for automatic SCSI identifier assignment.
3. Use the 50-pin, high-density shielded device connector defined in the SCSI-2 standard (external SCSI peripheral subsystems only).
4. Select at least three available DMA channels, either 8 bit or 16 bit, if DMA is supported.
5. Support disabling in case of resource conflicts with other devices.
6. Support automatic switchable termination for Plug and Play operation of internal, external, or mixed SCSI configurations.

SCSI Devices

SCSI® devices must meet the following requirements:

1. Meet the standards described in the current version of the Plug and Play SCSI specification.
2. Support the SCSI Configured Auto-Magically (SCAM) Level 1 protocol for automatic SCSI identifier assignment.
3. Use the 50-pin, high-density shielded device connector defined in the SCSI-2 standard (external SCSI peripheral subsystems only).
4. Use the drivers and receivers that meet the specifications defined in the single-ended alternative of the SPI.
5. Use cables that conform to the cable requirements defined in clause 6 of the SPI specification.
6. Ensure that external SCSI peripherals contain two connectors for the SCSI cable: a SCSI in connector and a SCSI out connector. The last peripheral in the chain uses a terminator on the SCSI out connector.
7. Support the attachment of a permanent terminator to the end of the cable for internal SCSI peripherals.
8. Ensure that internal SCSI peripherals do not terminate the SCSI bus.
9. Ensure that terminations conform to the terminator requirements in the SPI specification over the terminator power (TERMPWR) voltage range of 4.0 to 5.25 VDC.
10. Power terminators from the TERMPWR line on the SCSI bus.
11. Provide overcurrent protection for the TERMPWR line or lines.
12. Ensure that only terminators draw power from TERMPWR.
13. Implement the SCSI Bus Parity signal defined in the SCSI-2 specifications.

Parallel Port Devices

Parallel port devices (printers) must meet the following requirements:

1. Meet the standards described in the current version of the Plug and Play Parallel Port Device specification.
2. Comply with IEEE P1284-I.
3. Support the compatibility and nibble mode protocols to read the device identifier from the peripheral.

External Communications Devices

An external communications device must be able to identify itself using the identification method described in the Plug and Play External COM Device Specification.

Modems

Modems must meet the following requirements:

1. Support at least 9600 bits per second (bps) V.32 with V42/V42bis protocol for data modems.
2. Support the TIA-602 (Hayes-compatible) AT command set, with extensions for flow control, V42/V42bis.
3. Support fax capabilities of at least 9600 bps V.29 with class 1 (TIA-578A).
4. Support Plug and Play device identification, using the appropriate Plug and Play specification (for example, ISA bus, COM port, PCMCIA slot, or LPT port).
5. Support the 16550A compatible universal asynchronous receiver-transmitter (UART) interface.

Network Adapters

Network adapters must meet the following requirements:

1. Support the network driver interface specification (NDIS) 3.1 network device driver, which allows dynamic starting and stopping of the network card.
2. Provide a means of automatically enabling the adapter as a boot device or enabling the adapter as a nonbootable device, if the network adapter is designed with Remote Initial Program Load (RIPL) capability.
3. Do not hook Interrupt 18 and Interrupt 19 on ISA bus systems. This is a requirement for an ISA Plug and Play card.
4. Support at least seven IRQ signals and enable/disable.
5. Select at least three available DMA channels, either 8 bit or 16 bit, if DMA is supported.
6. Support disabling in case of resource conflicts with other devices.

The Generic Win32-based Application

This section introduces the source code components of a generic Win32-based application, named GENERIC. They closely resemble their counterparts in a generic 16-bit Windows-based application. This section assumes that you have used Win32-based applications and therefore are already familiar with windows, menus, and dialog boxes. The GENERIC application described here consists of the following parts:

- The entry-point function
- The menu
- The window procedure
- The About dialog box

The complete code is shown in Source Code.

The Entry-Point Function

Every Win32-based application must have an entry-point function. The name commonly given to the entry point is the **WinMain** function.
As in most Win32-based applications, the **WinMain** function for the GENERIC application completes the following steps:

- Registering the window class
- Creating the main window
- Entering the message loop

Registering the Window Class

Every window must have a window class. A window class defines the attributes of a window, such as its style, its icon, its cursor, the name of the menu, and the name of the window procedure.
The first step is to fill in a **WNDCLASS** structure with class information. Next, you pass the structures to the **RegisterClass** function. The GENERIC application registers the GenericAppClass window class as follows:

```
wc.lpszClassName = "GenericAppClass";
wc.lpfnWndProc = MainWndProc;
wc.style = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
wc.hInstance = hInstance;
wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
wc.hCursor = LoadCursor( NULL, IDC_ARROW );
wc.hbrBackground = (HBRUSH)( COLOR_WINDOW+1 );
wc.lpszMenuName = "GenericAppMenu";
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;

RegisterClass( &wc );
```

For more information on the menu, see The Menu. For more information on the window procedure, see The Window Procedure.

Creating the Main Window

You can create the window by calling the **CreateWindow** function. The GENERIC application creates the window as follows:

```
hWnd = CreateWindow( "GenericAppClass",
    "Generic Application",
    WS_OVERLAPPEDWINDOW|WS_HSCROLL|WS_VSCROLL,
    0,
    0,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
);
```

The first parameter is the name of the class that we registered. The remaining parameters specify other window attributes. This call creates the window, but Windows does not display a window until the application calls the **ShowWindow** function. The GENERIC application displays the window as follows:

```
ShowWindow( hWnd, nCmdShow );
```

Entering the Message Loop

Once the main window is created and displayed, the **WinMain** function can being its primary task, which is to read messages from the application queue and dispatch them to the appropriate window.
Windows does not send input directly to an application. Instead, it places all mouse and keyboard input from the user into a message queue, along with messages posted by Windows and other application. The application must read the message queue, retrieve the messages, and dispatch them so that the window procedure can process them.
The GENERIC application uses the following message loop:

```
while( GetMessage( &msg, NULL, 0, 0 ) ) {
   TranslateMessage( &msg );
   DispatchMessage( &msg );
}
```

The **GetMessage** function retrieves a message from the queue. The **DispatchMessage** function sends each message to the appropriate window procedure. The **TranslateMessage** function translates virtual-key message into character messages. In GENERIC, this is necessary to implement menu access keys.

The Menu

Most applications include a menu to provide a means for the user to select commands. The most common way to create a menu is to define it as a resource in the resource-definition file. The GENERIC application has a single menu, named Help, with a single command, About. The resource is defined as follows:

```
GenericAppMenu MENU
{
   POPUP "&Help"
   {
      MENUITEM "&About",          IDM_ABOUT
   }
}
```

The name of the menu resource is specified when registering the window class.
Selecting the About command causes GENERIC to display the about dialog box.

The Window Procedure

Every window must have a window procedure. The name of the window procedure is user-defined. The GENERIC application uses the following window procedure for the main window:

```
LRESULT WINAPI MainWndProc( HWND, UINT, WPARAM, LPARAM );
```

The WINAPI modifier is used because the window procedure must be declared with the standard call calling convention.
The window procedure receives messages from Windows. These may be input messages or window-management messages from Windows. You can optionally handle a message in your window procedure or pass the message to Windows for default processing by calling the **DefWindowProc** function. The GENERIC application processes the WM_PAINT, WM_COMMAND, and WM_DESTROY messages, using a switch statement that is structured as follows:

```
    switch( msg ) {
       case WM_PAINT:
       ...
       case WM_COMMAND:
       ...
       case WM_DESTORY:
       ...
       default:
          return( DefWindowProc( hWnd, msg, wParam, lParam ));
```

The WM_PAINT message indicates that you should redraw what's in all or part of your application's window. Use the **BeginPaint** function to get a handle to a device context, then use the device context for drawing within the application's window, with functions like **TextOut**. Use **EndPaint** to release the device context. The GENERIC application displays a text string, "Hello, World!", in the window using the following code:

```
    case WM_PAINT:
       hDC = BeginPaint( hWnd, &ps );

       TextOut( hDC, 10, 10, "Hello, World!", 13 );

       EndPaint( hWnd, &ps );
       break;
```

The WM_COMMAND message indicates that a user has selected a command item from a menu. The GENERIC application uses the following code to check if its About menu item has been selected:

```
       case WM_COMMAND:
          switch( wParam ) {
             case IDM_ABOUT:
             ...
             break;
          }
```

Most window procedures process the WM_DESTROY message. Windows sends this message to the window procedure immediately after destroying the window. The message gives you the opportunity to finish processing and post a WM_QUIT message in the application queue. The GENERIC application handles the WM_DESTROY message as follows:

```
case WM_DESTROY:
    PostQuitMessage( 0 );
    break;
```

The About Dialog Box

A dialog box is a temporary window that displays information or prompts the user for input. The GENERIC application includes an About dialog box. Every application should include an About dialog box. The dialog box displays such information as the application's name and copyright information.
You create and display a dialog box by using the **DialogBox** function. This function takes a dialog box template and creates a dialog box.
The GENERIC application includes the following dialog box template in the resource-definition file:

```
AboutDlg DIALOG FIXED 6, 21, 198, 99
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "About Generic"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON   "&OK", IDOK, 72, 74, 40, 14
    LTEXT           "Generic Application", 104, 45, 14,
                    128, 8
    LTEXT           "Written as a sample", 105, 45, 35, 59, 8
    LTEXT           "Microsoft Corporation", 106, 45, 45, 98, 8
    LTEXT           "Copyright (c) 1996", 107, 45,
                    54, 138, 8
END
```

The name of the source is specified as AboutDlg. For more information, see Dialog Resource.
When the user clicks About from the Help menu, the following code in the window procedure displays the About dialog box:

```
case WM_COMMAND:
   switch( wParam ) {
      case IDM_ABOUT:
         DialogBox( ghInstance, "AboutDlg", hWnd, (DLGPROC)
                    AboutDlgProc );
      break;
   }
break;
```

The last parameter is a pointer to a dialog box procedure. It has the following prototype.

```
LRESULT WINAPI AboutDlgProc( HWND, UINT, WPARAM, LPARAM );
```

A dialog box procedure is similar to a window procedure, but usually processes only dialog initialization and user-input messages. The GENERIC application contains the following message processing code:

```
switch( uMsg ) {
   case WM_INITDIALOG:
      return TRUE;
   case WM_COMMAND:
      switch( wParam ) {
         case IDOK:
            EndDialog( hDlg, TRUE );
            return TRUE;
      }
   break;
}

return FALSE;
```

Source Code

The GENERIC application consists of the following files:

- GENERIC.C
- GENERIC.H
- GENERIC.RC
- GENERIC.DLG

GENERIC.C

GENERIC.C contains code for the GENERIC application. It includes GENERIC.H.

```
/****************************************************************\
*  generic.c: Source code for generic                            *
*                                                                *
*  Comments: Generic Win32-based Application                     *
*                                                                *
*  Functions:                                                    *
*     WinMain      - Application entry point                     *
*     MainWndProc  - main window procedure                       *
*     AboutDlgProc - dialog procedure for About dialog           *
*                                                                *
*                                                                *
\****************************************************************/

/*********************  Header Files  *********************/

#include <windows.h>
#include "generic.h"

/*********************  Prototypes  ***********************/

LRESULT WINAPI MainWndProc( HWND, UINT, WPARAM, LPARAM );
LRESULT WINAPI AboutDlgProc( HWND, UINT, WPARAM, LPARAM );

/*******************  Global Variables ********************/

HANDLE ghInstance;

/****************************************************************\
*  Function: int PASCAL WinMain(HINSTANCE, HINSTANCE, LPSTR, int)*
*                                                                *
*   Purpose: Initializes Application                             *
*                                                                *
*  Comments: Register window class, create and display the main  *
*            window, and enter message loop.                     *
*                                                                *
*                                                                *
\****************************************************************/

int PASCAL WinMain( HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpszCmdLine,
    int nCmdShow )
{
   WNDCLASS wc;
   MSG msg;
   HWND hWnd;

   if( !hPrevInstance )
   {
      wc.lpszClassName = "GenericAppClass";
      wc.lpfnWndProc = MainWndProc;
```

```
      wc.style = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
      wc.hInstance = hInstance;
      wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
      wc.hCursor = LoadCursor( NULL, IDC_ARROW );
      wc.hbrBackground = (HBRUSH)( COLOR_WINDOW+1 );
      wc.lpszMenuName = "GenericAppMenu";
      wc.cbClsExtra = 0;
      wc.cbWndExtra = 0;

      RegisterClass( &wc );
   }

   ghInstance = hInstance;

   hWnd = CreateWindow( "GenericAppClass",
      "Generic Application",
      WS_OVERLAPPEDWINDOW|WS_HSCROLL|WS_VSCROLL,
      0,
      0,
      CW_USEDEFAULT,
      CW_USEDEFAULT,
      NULL,
      NULL,
      hInstance,
      NULL
   );

   ShowWindow( hWnd, nCmdShow );

   while( GetMessage( &msg, NULL, 0, 0 ) ) {
      TranslateMessage( &msg );
      DispatchMessage( &msg );
   }

   return msg.wParam;
}

/****************************************************************\
* Function: LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM) *
*                                                                *
*  Purpose: Processes Application Messages                       *
*                                                                *
* Comments: The following messages are processed                 *
*                                                                *
*           WM_PAINT                                             *
*           WM_COMMAND                                           *
*           WM_DESTROY                                           *
*                                                                *
*                                                                *
\****************************************************************/

LRESULT CALLBACK MainWndProc( HWND hWnd, UINT msg, WPARAM wParam,
   LPARAM lParam )
{
   PAINTSTRUCT ps;
```

```
   HDC hDC;

   switch( msg ) {

/****************************************************************\
*     WM_PAINT:                                                  *
\****************************************************************/

      case WM_PAINT:
         hDC = BeginPaint( hWnd, &ps );

         TextOut( hDC, 10, 10, "Hello, World!", 13 );

         EndPaint( hWnd, &ps );
         break;

/****************************************************************\
*     WM_COMMAND:                                                *
\****************************************************************/

      case WM_COMMAND:
         switch( wParam ) {
            case IDM_ABOUT:
               DialogBox( ghInstance, "AboutDlg", hWnd, (DLGPROC)
                          AboutDlgProc );
            break;
         }
      break;

/****************************************************************\
*     WM_DESTROY: PostQuitMessage() is called                    *
\****************************************************************/

      case WM_DESTROY:
         PostQuitMessage( 0 );
         break;

/****************************************************************\
*     Let the default window proc handle all other messages      *
\****************************************************************/

      default:
         return( DefWindowProc( hWnd, msg, wParam, lParam ));
   }

   return 0;
}

/****************************************************************************\
* Function: LRESULT CALLBACK AboutDlgProc(HWND, UINT, WPARAM, LPARAM)*
*                                                                            *
*  Purpose: Processes "About" Dialog Box Messages                            *
*                                                                            *
* Comments: The About dialog box is displayed when the user clicks           *
*           About from the Help menu.                                        *
```

```
*                                                                    *
\********************************************************************/

LRESULT CALLBACK AboutDlgProc( HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM
lParam )
{
   switch( uMsg ) {
      case WM_INITDIALOG:
         return TRUE;
      case WM_COMMAND:
         switch( wParam ) {
            case IDOK:
               EndDialog( hDlg, TRUE );
               return TRUE;
         }
      break;
   }

   return FALSE;
}
```

GENERIC.H

GENERIC.C contains header information for the GENERIC application. It is included in GENERIC.C and GENERIC.RC.

```
/************************************************************\
* generic.h: Header file for Generic                         *
*                                                            *
*                                                            *
\************************************************************/

/******* Menu Defines *******/

#define IDM_ABOUT      1000
```

GENERIC.RC

GENERIC.RC contains resource information for the GENERIC application. The dialog resources are included in GENERIC.DLG.

```
/********************************************************\
* generic.rc: Resource script for Generic                *
*                                                        *
*                                                        *
\********************************************************/

#include <windows.h>
#include "generic.h"
#include "generic.dlg"

GenericAppMenu MENU
{
   POPUP "&Help"
   {
      MENUITEM "&About",          IDM_ABOUT
   }
}
```

GENERIC.DLG

GENERIC.DLG defines the About dialog box for the GENERIC application. This file is included in GENERIC.RC.

```
/************************************************************\
* generic.dlg: Dialogs for Generic                           *
*                                                            *
*                                                            *
\************************************************************/

1 DLGINCLUDE "generic.h"

AboutDlg DIALOG FIXED 6, 21, 198, 99
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "About Generic"
FONT 8, "MS Shell Dlg"
BEGIN
    DEFPUSHBUTTON   "&OK", IDOK, 72, 74, 40, 14
    LTEXT           "Generic Application", 104, 45, 14,
                    128, 8
    LTEXT           "Written as a sample", 105, 45, 35, 59, 8
    LTEXT           "Microsoft Corporation", 106, 45, 45, 98, 8
    LTEXT           "Copyright (c) 1996", 107, 45,
                    54, 138, 8
END
```

Porting Code from 16-bit Windows to 32-bit Windows

This chapter describes how to create a 32-bit version of an application written for Windows 3.x, as well as how to make the code portable *between* versions of Windows. Portable code can be recompiled as either a 16-bit application or a 32-bit application.

This chapter includes the following sections:

Overview of Changes

The Win32 Application Program Interface (API) uses the flat 32-bit addressing mode, supports source code portability between different processors, and supports high-end capabilities such as security and nonpreemptive multitasking.
One of the major design goals of the Win32 API was to minimize the impact on existing code, so that 16-bit applications can be adapted as easily as possible. However, some changes were mandated by the larger address space. Items which have increased to 32 bits include the following:

- Pointers
- Window handles
- Handles to other objects, such as pens, brushes, and menus
- Message parameters
- Graphics coordinates

These size differences are generally resolved in the header files, but some changes your to source code are necessary. For example, the contents of some message parameters have changed, so you must rewrite the code that handles these messages. The larger size of graphics coordinates also affects a number of function calls.

Porting Your Application

Although the Win32 API was designed to be as compatible as possible, you may need to modify a large amount of source code. To make these changes effectively and efficiently, the following top-down approach is recommended:

1. Build the application with 32-bit development tools, and fix any errors.
2. Replace complex procedures that are difficult to port, as well as procedures written in assembly language, with stub procedures that do nothing except return.
3. Port the code in the main portion of the application, using the steps described in this chapter.
4. Fill in each of the stub procedures, one at a time, with portable code once the main portion of the application compiles and runs correctly.

The sections listed describe steps you should take each time you port a 16-bit Windows-based application to the Win32 API:

- Revising the Window Procedure Declaration
- Using Proper Data Types
- Handling Messages
- Adjusting Function Calls
- Revising the WinMain Function
- Revising Dynamic-Link Libraries

Revising the Window Procedure Declaration

The first step in porting a 16-bit Windows-based application to the Win32 API is to revise the declaration of the window procedure. The standard declaration of a window procedure for a 16-bit Windows-based application is:

```
LONG FAR PASCAL MainWndProc(
    HWND hWnd,
    unsigned message,
    WORD wParam,
    LONG lParam
);
```

Note the use of the **FAR PASCAL**, **unsigned**, and **WORD** modifiers. To revise the declaration, replace these modifiers as shown below:

```
LRESULT CALLBACK MainWndProc(
    HWND hWnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam
);
```

The following table summarizes the changes to the declaration:

Windows 3.x	Win32 (portable code)	Reason for changing
FAR PASCAL	**CALLBACK**	**CALLBACK** is guaranteed to use whatever calling convention is appropriate for Windows.
unsigned	**UINT**	Meaning is the same, but **UINT** guarantees portability.
WORD	**WPARAM**	**WORD** is always 16 bits. **WPARAM** is portable.
LONG	**LPARAM**	Not required, but consistent with **WPARAM** convention

The most significant difference between the Windows 3.x declaration and the Win32 API version is the size difference of the *wParam* parameter.

Using Proper Data Types

Source code written for 16-bit Windows often uses the type **WORD** interchangeably with types such as **HWND** and **HANDLE**. For example, the typecast **(WORD)** might be used to cast a data type to a handle:

```
hWnd = (WORD) SendMessage( hWnd, WM_GETMDIACTIVATE, 0, 0 );
```

This code compiles correctly with a 16-bit compiler, because both the **WORD** type and handles are 16 bits. However, the code does not compile correctly with a 32-bit compiler, because handles are 32 bits while the **WORD** type is still 16 bits.
To make this example portable, replace the **WORD** cast by and **HWND** cast, as shown:

```
hWnd = (HWND) SendMessage( hWnd, WM_GETMDIACTIVATE, 0, 0 );
```

To write portable code, examine each **WORD** cast and **WORD** definition in your code, and revise as follows:

- If a variable or expression is to hold a handle, replace **WORD** with the appropriate handle type, such as **HWND**, **HPEN**, or **HINSTANCE**. Avoid using a generic handle type.
- If a variable or expression is a graphics coordinate or some other integer value that has increased from 16 to 32 bits, replace **WORD** with **UINT**.
- Maintain use of the **WORD** type only if the data type needs to be 16 bits for all versions of Windows (usually because it is a function argument or structure member).

UINT is defined to be the natural integer size for both the 16-bit and 32-bit environments. The 32-bit platforms, especially RISC, handle 32-bit data more efficiently than 16-bit data.

Handling Messages

When you move from Windows 3.x to the Win32 API, the information packing changes for some messages. You can either:

- Revise the code so that it works only for the 32-bit version.
- Make the message-handling code portable, so that you can easily compile for either the 16-bit or 32-bit environment.

We strongly recommend that you create portable code. This section focuses primarily on writing portable versions of the code.

Handles are now 32 bits in the Win32 API and can therefore no longer be combined with other information and still fit into a 32 bit parameter (*lParam*), as was possible in Windows 3.x. The handle now occupies all of *lParam*, so information formerly in the high or low word of *lParam* must now move to *wParam*.

Because the *wParam* message parameter is now 32 bits in the Win32 API, it can hold the information that can no longer be held in *lParam*. The following table shows how this repacking works for WM_COMMAND, one of the messages affected:

Environment	wParam	lParam
Windows 3.x	**id** (16-bits)	**hwnd** (16 bits), **cmd** (16-bits)
Win32 API	**id** (16-bits), **cmd** (16-bits)	**hwnd** (32 bits)

Extracting Data from Messages with Portable Code

When your code handles a Windows message that has been repacked, the cleanest way to revise the code is to first extract needed information from the message and store the information in local variables. This localizes message packing issues to a few lines of your code.
For example, if your application was written for 16 bits, you can use the following code to handle the WM_COMMAND message:

```
case WM_COMMAND:
    id = wParam;
    hwndChild = LOWORD(lParam);
    cmd = HIWORD(lParam);
```

Here is the portable version of the same code, which yields correct results, whether you compile for the 16-bit or 32-bit environment:

```
case WM_COMMAND:
    id = LOWORD(wParam);
    hwndChild = (HWND)(UINT)lParam;
#ifdef WIN32
    cmd = HIWORD(wParam);
#else
    cmd = HIWORD(lParam);
#endif
```

This example illustrates these common repacking problems:

- Data always occupying the low 16 bits
- A handle changing size but not location
- Data that changes in location

Data Always Occupying the Low 16 Bits

The **id** data, in this case, always occupies the low word of *wParam* and is always 16 bits. The **LOWORD** macro produces the correct results because it always returns a 16-bit data type. The result is either all of *wParam* (if 16 bits) or the low half (if 32 bits):

```
id = LOWORD(wParam);        // 16-bit: id=wParam
```

Handles Not Changing Location

The **hwndChild** data is the same address as *lParam*; it is either in the low byte or it occupies all of *lParam*. As long as the address of a handle is always that of *lParam*, using the cast **(HWND)(UINT)** correctly extracts the handle:

```
hwndChild = (HWND)(UINT)lParam;        // 16-bit: hwnd=LOWORD(lParam);
```

Data Changing Location

In cases where a piece of data changes packing location in the Win32 API, you need to use an **#ifdef** statement or you can write your own conversion macros. In this case, the **cmd** data moves from the high word of *wParam* to the high word of *wParam*, so the portable version of the code is:

```
#ifdef WIN32
cmd = HIWORD(wParam);
#else
cmd = HIWORD(lParam);
#endif
```

Often, **LOWORD** and **HIWORD** macros create portability problems, because you should usually (except as indicated here) not be extracting parts of data types.

Using Message Crackers to Write Portable Code

Message crackers are a set of macros that parse message parameters. They enable you to write portable code without having to use **#ifdef** statements to parse messages. Some examples of macros used to parse message parameters follow:

```
GET_WM_COMMAND_ID  (wParam, lParam)   // Parse control ID value
GET_WM_COMMAND_HWND(wParam, lParam)   // Parse control HWND
GET_WM_COMMAND_CMD (wParam, lParam)   // Parse notification command
```

Each platform has a WINDOWSX.H header file that defines these macros as appropriate. Refer to Handling Messages with Portable Macros for more information.

Summary of Windows Messages Affected

Use the following table to reference the packing of Windows messages affected by porting. The approaches discussed in the previous sections can be used for each message. Where two parameters are given, the one listed first corresponds to the least-significant 16 bits.
Except for the WM_CTLCOLOR messages, each message is given below with two rows: the first row gives 16-bit Windows packing for the message, the one below it gives Win32 API packing.

Message		wParam: Windows 3.x Win32 API	lParam: Windows 3.x Win32 API
WM_ACTIVATE	(16-bit Windows)	**state**	**fMinimized**, **hwnd**
	(Win32 API)	**state**, **fMinimized**	**hwnd** (32 bits)
WM_CHARTOITEM	(16-bit Windows)	**char**	**pos**, **hwnd**
	(Win32 API)	**char**, **pos**	**hwnd** (32 bits)
WM_COMMAND	(16-bit Windows)	**id**	**hwnd**, **cmd**
	(Win32 API)	**id**, **cmd**	**hwnd** (32 bits)
WM_CTLCOLOR	(16-bit Windows)	**hdc**	**hwnd**, **type**
WM_CTLCOLOR*type* (Win32 API)	(Win32 API)	**hdc** (32 bits)	**hwnd** (32 bits)

Note In the Win32 API, WM_CTLCOLOR is replaced by a series of messages, each corresponding to a different type. To write portable code, use **#ifdef** to handle this difference.

Message		wParam: Windows 3.x Win32 API	lParam: Windows 3.x Win32 API
WM_MENUSELECT	(16-bit Windows)	**cmd**	**flags**, **hMenu**
	(Win32 API)	**cmd**, **flags**	**hMenu** (32 bits)
WM_MDIACTIVATE	(16-bit Windows)	**fActivate**	**hwndDeactivate**, **hwndActivate**
	(Win32 API)	**hwndActivate** (32 bits)	**hwndDeactivate** (32 bits)
WM_MDISETMENU	(16-bit Windows)	0	**hMenuFrame**, **hMenuWindow**
	(Win32 API)	**hMenuFrame** (32 bits)	**hMenuWindow** (32 bits)
WM_MENUCHAR	(16-bit Windows)	**char**	**hMenu**, **fMenu**
	(Win32 API)	**char**, **fMenu**	**hMenu** (32 bits)
WM_PARENTNOTIFY	(16-bit Windows)	**msg**	**id**, **hwndChild**
	(Win32 API)	**msg**, **id**	**hwndChild** (32 bits)

WM_VKEYTOITEM	(16-bit Windows)	**code**	**item**, **hwnd**
	(Win32 API)	**code**, **item**	**hwnd** (32 bits)
EM_GETSEL (returns wStart, wEnd)	(16-bit Windows)	0	0
	(Win32 API)	0 or **lpdwStart**	0 or **lpdwEnd**
EM_LINESCROLL	(16-bit Windows)	0	**nLinesVert**, **nLinesHorz**
	(Win32 API)	**mLinesHorz** (32 bits)	**nLinesVert** (32 bits)
EM_SETSEL	(16-bit Windows)	0	**wStart**, **wEnd**
	(Win32 API)	**wStart** (32 bits)	**wEnd** (32 bits)
WM_HSCROLL, WM_VSCROLL	(16-bit Windows)	**code**	**pos**, **hwnd**
	(Win32 API)	**code**, **pos**	**hwnd** (32 bits)

Summary of DDE Messages Affected

DDE messages are packed differently for the Win32 API and Windows 3.x. These differences are shown in the following table.

Message		wParam: Windows 3.x Win32 API	lParam: Windows 3.x Win32 API
WM_DDE_ACK (posted form only)	(16-bit Windows)	**hwnd**	**wStatus**, **aItem** or **wStatus**, **hCommands**
	(Win32 API)	**hwnd** (32 bits)	**hDDEAck** (see below)
WM_DDE_ADVISE	(16-bit Windows)	**hwnd**	**hOptions**, **aItem**
	(Win32 API)	**hwnd** (32 bits)	**hDDEAdvise** (see below)
WM_DDE_DATA	(16-bit Windows)	**hwnd**	**hData**, **aItem**
	(Win32 API)	**hwnd** (32 bits)	**hDDEData** (see below)
WM_DDE_POKE	(16-bit Windows)	**hwnd**	**hData**, **aItem**
	(Win32 API)	**hwnd** (32 bits)	**hDDEPoke** (see below)

Because of storage limitations, some of the information in the 32-bit versions of the messages is stored in a structure, which is accessed through the handle in *lParam*. You can use the following functions to extract information from these structures:

PackDDElParam
UnPackDDElParam
FreeDDElParam

Adjusting Function Calls

You may need to revise code if you call functions in any of the following categories:

- Graphics functions
- Functions accessing "extra" window data
- MS-DOS system calls
- Far-pointer functions
- Functions getting list and combo box contents

Graphics Functions

Most of the Windows 3.x functions that must be replaced return packed x- and y-coordinates.
In Windows 3.x, the x- and y-coordinates are 16 bits each and are packed into the 32-bit **DWORD** function return value, the largest valid size. In the Win32 API, the coordinates are 32 bits each, totaling 64 bits, and are thus too large to fit into a single return value. Each Windows 3.x function is replaced by a function with the same name, but with an **Ex** suffix added. The **Ex** functions pass the x- and y-coordinates using an additional parameter instead of a return value. These new functions are supported by both the Win32 API and Windows 3.x.
Windows 3.x implements these functions with a static library, in order that source code compiled with the new calls will also function in Windows 3.x.
The problematic graphics functions fall into two groups. The first group, functions that set coordinates, are shown below with the Win32 API versions:

Windows 3.x function	Win32 API version of function
MoveTo	**MoveToEx**
OffsetViewportOrg	**OffsetViewportOrgEx**
OffsetWindowOrg	**OffsetWindowOrgEx**
ScaleViewportExt	**ScaleViewportExtEx**
ScaleWindowExt	**ScaleWindowExtEx**
SetBitmapDimension	**SetBitmapDimensionEx**
SetMetaFileBits	**SetMetaFileBitsEx**
SetViewportExt	**SetViewportExtEx**
SetWindowExt	**SetWindowExtEx**
SetWindowOrg	**SetWindowOrgEx**

Each of the functions in the first column returns a value, although application code frequently ignores it. However, even if you do not care about the return value, you must still replace the old function call by the new form. The old functions are not supported in the Win32 API.
Each **Ex** function includes an additional parameter that points to a location to receive data. After the function call, this data provides the same information as the corresponding function's return value. If you do not need this information, you can pass **NULL** to this parameter.
Under Windows 3.x, a call to the **MoveTo** function can be written as follows:

```
MoveTo( hDC, x, y );
```

In the portable version supported by all versions of Windows, the call to **MoveTo** is rewritten as follows. Note that the information returned by **MoveTo** under Windows 3.x is still ignored:

```
MoveToEx( hDC, x, y, NULL );
```

As a general rule, pass **NULL** as the last parameter unless you need to use the x- and y-coordinates returned by the Windows 3.x version. In the latter case, use the procedure outlined in the next few paragraphs.
The second group of functions consists of functions in which the application code normally does use the return value. They are listed in the following table.

Windows 3.x function	Portable version of function
GetAspectRatioFilter	**GetAspectRatioFilterEx**
GetBitmapDimension	**GetBitmapDimensionEx**
GetBrushOrg	**GetBrushOrgEx**
GetCurrentPosition	**GetCurrentPositionEx**
GetTextExtent	**GetTextExtentPoint**
GetTextExtentEx	**GetTextExtentExPoint**
GetViewportExt	**GetViewportExtEx**
GetViewportOrg	**GetViewportOrgEx**
GetWindowExt	**GetWindowExtEx**

GetWindowOrg **GetWindowOrgEx**

The **GetTextExtent** function uses the **Point** suffix, because there is already a Windows 3.1 extended function **GetTextExtentEx**. Therefore, the **Point** suffix is added to the functions **GetTextExtent** and **GetTextExtentEx**, to name the portable versions for each.
As with the first group of functions, the **Ex** (and **Point**) versions each add an additional parameter: a pointer to a **POINT** or **SIZE** structure to receive x/y coordinates. Because this structure is always the appropriate size for the environment, so you can write portable code by:

- Declaring a local variable of type **POINT** or **SIZE**, as appropriate.
- Passing a pointer to this structure as the last parameter to the function.
- Calling the function. The function responds by filling the structure with the appropriate information.

For example, the Windows 3.x version call to **GetTextExtent** extracts the x- and y-coordinates from a **DWORD** return value (stored in a temporary variable, dwXY):

```
DWORD dwXY;

dwXY = GetTextExtent( hDC, szLabel1, strlen( szFoo ) );
rect.left = 0; rect.bottom = 0;
rect.right = LOWORD(dwXY);
rect.top = HIWORD(dwXY);
InvertRect( hDC, &rect );
```

The portable version passes a pointer to a temporary **SIZE** structure, and then it extracts data from the structure:

```
SIZE sizeRect;

GetTextExtentPoint( hDC, szLabel1, strlen( szLabel1 ), &sizeRect );
rect.left = 0; rect.bottom = 0;
rect.right = sizeRect.cx;
rect.top = sizeRect.cy;
InvertRect( hDC, &rect );
```

Functions That Access the Extra Window Data

The functions described in this section manipulate the "extra" data area of a window structure. This structure can contain system information as well as user-defined data. You specify the size of this data area by using the **cbClsExtra** member of the **WNDCLASS** structure when you register the window class.
The following Windows 3.x functions get or set 16 bits during each call:
GetClassWord
GetWindowWord
SetClassWord
SetWindowWord
You can use these functions in Windows 3.x to access system information, stored as 16-bit items. But in the Win32 API, each of these system-information items grows to 32 bits. Therefore, in a Win32-based application, you would use the following functions which access 32 bits at a time:
GetClassLong
GetWindowLong
SetClassLong
SetWindowLong
Each of these functions take two parameters: a window handle and an offset into the data area. These offsets differ depending on whether you are compiling for Windows 3.x or the Win32 API.
The index values specifying these offsets correspond to each other as follows. Note that neither version is portable.

Windows 3.x	**Win32 API (not portable)**
GCW_CURSOR	GCL_CURSOR
GCW_HBRBACKGROUND	GCL_HBRBACKGROUND
GCW_HICON	GCL_HICON
GWW_HINSTANCE	GWL_HINSTANCE
GWW_HWNDPARENT	GWL_HWNDPARENT
GWW_ID	GWL_ID
GWW_USERDATA	GWL_USERDATA

To create portable code using these offsets, you need to use **#ifdef** statements as shown below. Both the function and the value of the second parameter change:

```
#ifdef WIN32
hwndParent = (HWND)GetWindowLong(hWnd, GWL_HWNDPARENT);
#else
hwndParent = (HWND)GetWindowWord(hWnd, GWW_HWNDPARENT);
#endif
```

In the case of **GWW_HWNDPARENT**, you can avoid calls to **GetWindowLong** and **GetWindowWord** altogether, and instead use a single call to a new function, **GetParent**. This function returns a handle of the appropriate size. The following example illustrates a call to **GetParent** that has the same results as the **#ifdef** statements shown in the previous example:

```
hwndParent = GetParent( hWnd );
```

Remember that offsets may change for private data that you store in the Window structure. You should review this code carefully and recalculate offsets for Win32-based applications, noting that some data types, such as handles, increase in size.

Porting MS-DOS System Calls

The **DOS3Call** function in Windows 3.0 must be called from assembly language. It is typically used to perform file I/O. In a Win32-based application, assembly language code that calls **DOS3Call** should be replaced by the appropriate file I/O calls. Other (non-file) INT 21H functions should be replaced, as shown in the following table, with the portable Windows call:

INT 21H subfunction	MS-DOS operation	Win32 API equivalent
0EH	Select Disk	**SetCurrentDirectory**
19H	Get Current Disk	**GetCurrentDirectory**
2AH	Get Date	**GetSystemTime**
2BH	Set Date	**SetSystemTime**
2CH	Get Time	**GetSystemTime**
2DH	Set Time	**SetSystemTime**
36H	Get Disk Free Space	**GetDiskFreeSpace**
39H	Create Directory	**CreateDirectory**
3AH	Remove Directory	**RemoveDirectory**
3BH	Set Current Directory	**SetCurrentDirectory**
3CH	Create Handle	**CreateFile**
3DH	Open Handle	**CreateFile**
3EH	Close Handle	**CloseHandle**
3FH	Read Handle	**ReadFile**
40H	Write Handle	**WriteFile**
41H	Delete File	**DeleteFile**
42H	Move File Pointer	**SetFilePointer**
43H	Get File Attributes	**GetFileAttributes**
43H	Set File Attributes	**SetFileAttributes**
47H	Get Current Directory	**GetCurrentDirectory**
4EH	Find First File	**FindFirstFile**
4FH	Find Next File	**FindNextFile**
56H	Change Directory Entry	**MoveFile**
57H	Get Date/Time of File	**GetFileTime**
57H	Set Date/Time of File	**SetFileTime**
59H	Get Extended Error	**GetLastError**
5AH	Create Unique File	**GetTempFileName**
5BH	Create New File	**CreateFile**
5CH	Lock	**LockFile**
5CH	Unlock	**UnlockFile**
67H	Set Handle Count	**SetHandleCount**

File Operations

Fixed-length buffers for filenames and environment strings may need to be increased in size. Windows NT and Windows 95 support long filenames, rather than the 8.3 format supported by MS-DOS. You can make code more portable by allocating longer buffers or by using dynamic memory allocation. If you want to conserve memory under Windows 3.x, use **#ifdef** statements to allocate buffers of the proper length for the environment.

Another area in which you might need to make changes is low-level file I/O. In porting Windows 3.x code, some developers have chosen to change from using the Windows API file I/O functions (such as **_lopen** and **_lread**) to using the C run-time low-level I/O functions (such as **_open** and **_read**). All versions of the Windows API support binary mode only, not text mode, but the C run-time calls use text mode by default. Therefore, when changing from the Windows file I/O to the C run-time versions, open files in binary mode by doing one of the following:

- Link with BINMODE.OBJ, which changes the default mode for all file-open operations.
- Open the individual files with _O_BINARY flag set.
- Use **setmode** to change an open file to _O_BINARY.

Far-Pointer Functions

Windows 3.x provides functions for memory and file manipulation using far pointers, which have the form **_f***xxxx*. In the Win32 API, these functions are replaced by similarly named functions of the form *xxxx*, because there is no need for far pointers in Win32-based applications. In other words, the **_f** prefix was dropped from the name.
The 32-bit WINDOWSX.H file defines the **_f***xxxx* function names so that the **_f***xxxx* function names are equated to the corresponding functions that are still supported. This means that as long as you include WINDOWSX.H, you don't have to rewrite calls to these functions. Some of the definitions are:

```
#define _fmemcpy(x,y,z)     memcpy(x,y,z)
#define _fstrcpy(x,y)       strcpy(x,y)
#define _fstrcmp(x,y)       strcmp(x,y)
#define _fstrcat(x,y)       strcat(x,y)
```

Functions Getting List and Combo Box Contents

The Win32 API contains two new functions, shown in the following table, that provide an improved means of extracting list and combo box contents. In each case, the portable version of the function lets you specify a buffer size for a string that receives the information.

Windows 3.x function	Portable version of the function
DlgDirSelect	**DlgDirSelectEx**
DlgDirSelectComboBox	**DlgDirSelectComboBoxEx**

For example, Windows 3.x code might contain the following function call:

```
DlgDirSelect( hDlg, lpString, nIDListBox );
```

This line of code should be replaced by the following call to **DlgDirSelectEx**:

```
DlgDirSelectEx( hDlg, lpString, sizeof(lpString), nIDListBox );
```

Revising the WinMain Function

The parameter list for **WinMain** is the same for the Win32 API and Windows 3.x:

```
int PASCAL WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
);
```

However, you will need to revise the **WinMain** function if either of the following conditions is true:

- You need to know when another instance of the application is running.
- You you need to access the command line arguments.

This is due to are two differences in the values passed through these parameters:

- Unlike Windows 3.x, *hPrevInstance* always returns NULL.
- The *lpCmdLine* parameter points to a string containing the entire command line, not just the parameters.

For more information, see the next two sections:

- Initializing Instances
- Accessing the Command Line Through lpCmdLine

Initializing Instances

The *hPrevInstance* parameter always returns NULL in the Win32 API. This causes each instance of an application to act as though it were the only instance running. The application must register the window class, and it cannot access data used by other instances, except through standard interprocess communication techniques such as shared memory or DDE. Calls to **GetInstanceData** must be replaced with one of these techniques.
Source code for Windows 3.x normally tests *hPrevInstance* to see if another instance of the application is already running; if the value is NULL this indicates there is no previous instance, so the code registers the window class. This code is automatically portable and needs no change: the result in a Win32-based application is that it always registers the window class, which is correct behavior.
Some applications must know if other instances are running. Sometimes this is because data sharing is required. More frequently, it is because only one instance of the application should run at a time.
Applications cannot use *hPrevInstance* to test for previous instances in a Win32-based application. An alternative method must be used to locate a previous instance, such as:

- Creating and testing for the existence of a named mutex.
- Creating a unique named pipe
- Calling **FindWindow** with the window class and name.

Because a second instance of the application could be started and execute a call to FindWindow before the first instance has created its window, it is better to use a named object.

Accessing the Command Line Through lpCmdLine

In Windows 3.x, the *lpCmdLine* parameter points to a string containing the command line, starting with the first parameter. The name of the application is not included. In the Win32 API, *lpCmdLine* points to a string containing the entire command line, including the application itself.
To get the arguments in Unicode, call the **GetCommandLine** function.

Revising Dynamic-Link Libraries

You will need to make some changes to port a dynamic-link library (DLL) from Windows 3.x to 32-bits. There are several differences in the initialization and termination routines:

- In Win32-based DLLs, one function handles both initialization and termination. In Windows 3.x, the initialization function must be provided, and the termination function, if present, must be named WEP.
- In Win32-based DLLs, the initialization function is called every time a new process loads the DLL or an additional thread is created, and when a process or thread is terminated. In Windows 3.x, initialization and termination functions are called only once during the lifetime of the DLL.
- A Win32-based DLL can be written entirely in C, which is recommended as an aid to porting to other processors. In Windows 3.x, the start-up code is linked to an object file written in assembly language. With Microsoft development tools, you link in an object file, LIBENTRY.OBJ, that accesses information in information in registers and calls **LibMain** in the C code. Win32-based DLLs do not need this file.

The initialization function in Window 3.x has the following prototype:

```
int CALLBACK LibMain(
    HINSTANCE hinst,    // handle of library instance
    WORD wDataSeg,      // library data segment
    WORD cbHeapSize,    // default heap size
    LPSTR lpszCmdLine   // command-line arguments
);
```

The initialization function in a Win32-based DLL includes the hinst parameter, but does not include the other parameters, for the reasons described in the following table.

Parameter	Comment
wDataSeg	Not needed in Win32-based applications; memory model is flat, not segmented.
cbHeapSize	All calls to local memory management functions operate on the default heap
lpszCmdLine	The command line can be obtained through a call to the **GetCommandLine** function.

The initialization function in the Win32 API, **DllEntryPoint**, has the following prototype:

```
BOOL APIENTRY DllEntryPoint(
    HINSTANCE hInstDLL,  // handle to DLL module
    DWORD fdwReason,     // reason for calling function
    LPVOID lpvReserved   // reserved
);
```

The *fdwReason* parameter indicates whether initialization or termination is taking place, and whether a process or a thread is involved. For example, when a process first loads a DLL, **DllEntryPoint** is called with a DLL_PROCESS_ATTACH notification: it is assumed that the process has one thread to begin with. When additional threads are created, the function is called with a DLL_THREAD_ATTACH notification. The body of your **DllEntryPoint** should contain a statement similar to this one:

```
switch( dwReason)
{
    case DLL_PROCESS_ATTACH:
       ...
    case DLL_THREAD_ATTACH:
       ...
    case DLL_THREAD_DETACH:
       ...
    case DLL_PROCESS_DETACH:
```

```
        . . .
}
```

Your initialization function should return 1 to indicate success. Returning NULL indicates failure.
You do not need a WEP function.

Using PORTTOOL to Automate Porting

You can use the PORTTOOL utility (PORTTOOL.EXE) to help port applications more easily. This utility finds locations in your code, such as references to certain functions and messages, that are likely to need revision. You should use PORTTOOL in conjunction with the information in this overview.

PORTTOOL uses settings in the file PORT.INI to determine what items to look for. This file is based on the table in Obsolete Programming Elements.

Run PORTTOOL and load a Windows 3.x source file. Select the SearchAPI option from the Search menu to search for occurrences of problematic functions and messages. When an occurrence of either is found, a dialog box appears specifying the message or function and briefly suggesting what change needs to be made. Although the porting tool is not intended to replace your primary editor, it does support basic editing capabilities, including Cut, Paste, and Search.

Additional Porting Work

If you have applied the guidelines and procedures described in the preceding sections, you may well have been able to port your entire application. However, your application may require additional revision. You should scan through the following sections to make sure that your application does not need additional changes:

- Profile Strings and .INI Files
- Focus, Mouse Capture, and Localized Input
- Shared Graphical Objects
- Memory and Pointers
- Structure Alignment
- Ranges and Promotions

Profile Strings and .INI Files

Although all 32-bit Microsoft platforms support the Win32 API, not all of them support all the features. Windows NT supports some additional features not present in Windows 95 or Win32s and Windows 95 supports additional features not present in Win32s.

Windows 3.x applications write to .INI files to store application information. Win32 platforms use the registration database instead of .INI files. This database offers a number of advantages, including security controls that prevent an application from corrupting system information, error logging, remote software updating, and remote administration of workstation software.

You can write portable code by using the profile API supported by Windows 3.x and the Win32 API. Call the **GetProfileString** and **WriteProfileString** functions instead of accessing .INI files directly. These functions use whichever underlying mechanism (.INI file or registration database) is supported by the platform.

Focus, Mouse Capture, and Localized Input

Windows NT and Windows 95 differ from Windows 3.x in that each thread of execution has its own message queue for localized input. This change affects window focus and mouse capture.

Window Focus

In Windows NT and Windows 95, each thread of execution can set or get the focus only to windows created by the current thread. This behavior prevents applications from interfering with each other. One application's delay in responding cannot cause other applications to suspend their response to user actions, as often happens in Windows 3.x.

Consequently, the following functions work differently in the Win32 API:

GetActiveWindow
GetCapture
GetFocus
ReleaseCapture
SetActiveWindow
SetCapture
SetFocus

The Get functions can now return NULL, which could not happen in Windows 3.x. Therefore, it's important to test the return value of **GetFocus** before using it. Instead of returning the window handle of another thread, the function returns NULL. For example, you call **GetFocus** and another thread has the focus. Note that it's possible for a call to **GetFocus** to return NULL even though an earlier call to **SetFocus** successfully set the focus. Simliar considerations apply to **GetCapture** and **GetActiveWindow**.

The Set functions can only specify a window created by the current thread. If you attempt to pass a window handle created by another thread, the call to the Set function fails.

Mouse Capture

Mouse capture is also affected by localized input queues. If the mouse is captured on mouse down, the window capturing the mouse receives mouse input until the mouse button is released, as in Windows 3.x. But if the mouse is captured while the mouse button is up, the window receives mouse input only as long as the mouse is over that window or another window created by the same thread.

Shared Graphical Objects

Win32-based applications run in separate virtual address spaces. Graphical objects are specific to the application and cannot be manipulated by other processes as in Windows 3.x. A handle to a bitmap passed to another process cannot be used because the original process retains ownership.
Each process should create its own pens and brushes. A cooperative process may access the bitmap data in shared memory and create its own copy of the bitmap. Alterations to the bitmap must be communicated between the cooperative processes by way of interprocess communication.

Memory and Pointers

To be portable, source code must avoid any techniques that rely on the 16-bit *segment*:*offset* address structure, because all pointers used by Win32-based applications are 32 bits in size and use flat rather than segmented memory. This difference in pointer structure is usually not a problem unless the code uses **HIWORD**, **LOWORD**, or similar macros to manipulate portions of the pointer.

For example, in Windows 3.x, memory is allocated to align on a segment boundary, which makes memory allocation functions return a pointer with an offset of `0x0000`. The following code exploits this fact to run successfully under Windows 3.x:

```
ptr2 = ptr1 = malloc();                 // ptr2 = xxxx:0000
LOWORD( ptr2 ) = index * elementsize;   // Place offset of array element
                                        // into ptr2 low word
```

Such code does not work properly in a Win32-based application. But standard pointer constructs, such as the following, always result in portable code:

```
ptr1 = malloc();            // Set ptr1 to start of memory block
ptr2 = ptr1[i];             // Place offset of array element
```

Here are some other guidelines for dealing with pointers:

- All pointers, including those that access the local heap, are 32 bits in a Win32-based application.
- Addresses never wrap, as they can with the low word in segmented addressing; for example, in Windows 3.x, an address can wrap from 1000:FFFF to 1000:0000.
- Structures that hold near pointers in Windows 3.x must be revised because all pointers are 32 bits in a Win32-based application. This may affect code that uses constants to access structure members, and it may also affect alignment.

Structure Alignment

Applications should generally align structure members at addresses that are "natural" for the data type and the processor involved. For example, a 4-byte data member should have an address that is a multiple of four.
This principle is especially important when you write code for porting to multiple processors. A misaligned 4-byte data member, which is on an address that is not a multiple of four, causes a performance penalty with an 80386 processor and a hardware exception with a RISC processor. In the latter cases, although the exception is handled by the system, the performance penalty is significantly greater.
Alignment problems can be avoided by setting compiler options or adjusting your structure definitions to meet the alignment requirements. Use the following guidelines to ensure proper alignment for all processors:

Type	**Alignment**
char	Align on byte boundaries
short (16-bit)	Align on even byte boundaries
int and **long** (32-bit)	Align on 32-bit boundaries
float	Align on 32-bit boundaries
double	Align on 64-bit boundaries
structures	Align on 32-bit boundaries

Ranges and Promotions

Occurrences of **int**, **unsigned**, and **unsigned int** indicate potential portability problems because size and range are not constant. Data that would not exceed its range in a Win32-based application could exceed range in Windows 3.x. Sign extension also works differently, so exercise caution in performing bitwise manipulation of this data.
Source code that relies on wrapping often presents portability problems, and should be avoided. For example, a loop should not rely on an **unsigned int** variable wrapping at 65535 (the maximum value in Windows 3.x) back down to 0.

Handling Messages with Portable Macros

Instead of taking the case-by-case approach, you can use message-cracking macros to write message handlers similar to those you'd write when using Microsoft Foundation Classes. These message handlers use the same parameter list regardless of operating system, thereby solving message-packing issues. This guide describes these and other macros defined in WINDOWSX.H (or WINDOWSX.H16 in the case of 16-bit applications).
This guide includes the following topics:

Using Message Crackers

Message crackers are a set of macros that extract useful information from the *wParam* and *lParam* parameters of a message and hide the details of how information is packed.
Using message crackers initially requires you to revise some of your code. They also have a minor impact on performance by involving an additional function call. However, they offer the following major advantages:

- Portability. Message crackers free you from packing issues and guarantee proper extraction of information, regardless of which environment you're compiling for.
- Readability. With message crackers, you can understand source code because message parameters are translated into data with meaningful names.
- Ease of use. In addition to decoding *wParam* and *lParam*, message crackers place message-handling code in separate functions. Instead of a long **switch** statement, you have a separate handler for each message.

Overview of Message Crackers

You use message crackers in your code by writing a separate message handler function for each message. Then you use a macro to call each of those functions from within your window procedure.
Use of message crackers for all messages is recommended, but you can optionally combine code that uses message crackers for some messages with code that responds to other messages directly.

Note To use message crackers, make sure you include the file WINDOWSX.H (or WINDOWSX.H16, in the case of 16-bit applications).

Suppose you have a message, WM_THIS. The code to handle this message would look something like this:

```
LONG WINAPI My_WndProc( HWND hwnd, UINT msg, UINT wParam, LONG lParam )
.
.
.
switch( msg ) {
     case WM_THIS:
           // Place code to handle message here
```

To use message crackers, write a message handler, and then call it from the **switch** statement. Suppose that there are two pieces of information contained in the WM_THIS message: *thisHdc* and *thisData*. Message crackers unpack this information from *wParam* and *lParam*, and pass it as parameters to your message handler, MyWnd_OnThis:

```
switch ( msg) {
     case WM_THIS:
           return HANDLE_WM_THIS ( hwnd, wParam, lParam, MyWnd_OnThis );
...
}


LRESULT MyWnd_OnThis ( HWND hwnd, HDC thisHdc, WORD thisData )
{
     // Place code to handle message here
}
```

The parameters to MyWnd_OnThis (after *hwnd*, which is always the first parameter) consist of information directly usable by your code: *thisHdc* and *thisData*. The macro **HANDLE_WM_THIS** translates *wParam* and *lParam* into *thisHdc* and *thisData* as it makes the function call.
The following general steps summarize how to use message crackers:

1. Declare a prototype for each message-handling function.
2. In the windows procedure, call the message handler. Use either a message decoder (such as **HANDLE_WM_CREATE**) or the **HANDLE_MSG** macro.
3. Write the message handler. Use a message forwarder such as **FORWARD_WM_CREATE** to call the default message procedure.

Declaring Message-Handler Prototypes

To use message crackers, first declare a prototype for the message handling function ("message handler" for short). Although you can give your message handlers any name you want, a recommended convention is:
*WndClass*_**On***Msg*
in which *WndClass* is the name of the window class, and *Msg* is the name of the message in mixed case, with the "WM" dropped. For example, the following code contains prototypes for functions handling WM_CREATE, WM_PAINT, and WM_MOUSEMOVE:

```
BOOL MyWnd_OnCreate( HWND hwnd, CREATESTRUCT FAR* lpCreateStruct );
void MyWnd_OnPaint( HWND hwnd );
void MyWnd_OnMouseMove( HWND hwnd, int x, int y, UINT keyFlags );
```

The first parameter to each function is always *hwnd*, which is a handle to the window that received the message. The rest of the parameters vary; each message handler has its own customized parameter list. To declare the appropriate parameters for a message, see the corresponding definitions in WINDOWSX.H.

Calling the Message Handler

In your window procedure, you call a message handler by using a message-decoder macro such as **HANDLE_WM_CREATE** or **HANDLE_WM_PAINT**. The general form for using these macros is:

case *msg***:**

return HANDLE_*msg* **(hwnd, wParam, lParam,** *handler* **);**

You should always return the value of the macro, even if no return value is expected and the corresponding message handler has **void** return type.

For example, you could place the following macros in your code:

```
switch( msg ) {
      case WM_CREATE:
            return HANDLE_WM_CREATE( hwnd, wParam, lParam, MyWnd_OnCreate );
      case WM_PAINT:
            return HANDLE_WM_PAINT( hwnd, wParam, lParam, MyWnd_OnPaint );
      case WM_MOUSEMOVE:
            return HANDLE_WM_MOUSEMOVE( hwnd, wParam, lParam,
                                        MyWnd_OnMouseMove );
      .
      .
      .
```

Alternatively, you can use the generic **HANDLE_MSG** macro, which generates the same code as the previous example, but saves space:

```
switch( msg ) {
      HANDLE_MSG( hwnd, WM_CREATE, MyWnd_OnCreate );
      HANDLE_MSG( hwnd, WM_PAINT, MyWnd_OnPaint );
      HANDLE_MSG( hwnd, WM_MOUSEMOVE, MyWnd_OnMouseMove );
      .
      .
      .
```

HANDLE_MSG assumes that you use the names *wParam* and *lParam* in the window procedure parameter list. You cannot use this macro if you have given these parameters other names.

Writing the Message Handler

In the message-handling function, you respond to the message using parameters that have been translated from *wParam* and *lParam* and passed to the function. In the following example, *lpCreateStruct* is an example of a parameter translated from *wParam* and *lParam*:

```
BOOL MyCls_OnCreate(HWND hwnd, CREATESTRUCT FAR* lpCreateStruct)
{
     // Place message-handling code here

    return FORWARD_WM_CREATE(hwnd, lpCreateStruct, DefWindowProc);
}
```

Message-handling code often finishes by calling **DefWindowProc** or some other default message procedure. You make this function call by using a "message forwarder," which uses the following form:
return FORWARD_*msg***(** *parmlist***,** *defaultMsgProc* **);**
The *parmlist* is the same list of parameters in the message handler, and *defaultMsgProc* is the default message procedure, typically **DefWindowProc**. The message forwarder repacks the information in the parameter list into the appropriate *wParam*/*lParam* format (depending on target environment) and forwards the message to the default message procedure.

Putting It Together: An Example

The following example demonstrates the use of several message handlers in a window procedure, showing where the various prototypes and macros fit into the code.
The header file, MYAPP.H, consists of function prototypes, including prototypes for the message handlers. Note how each message handler has its own parameter list, which is customized to best represent the information packed in the corresponding message:

```
// MYAPP.H

// Window procedure prototype

LONG WINAPI MyWnd_WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam);

// Default message handler

#define MyWnd_DefProc    DefWindowProc

// MyWnd class message handler functions, declared in a .h file:
//
void MyWnd_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags);
void MyWnd_OnLButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT
keyFlags);
void MyWnd_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags);
```

The rest of the code in this example is in MYAPP.C, which contains the window procedure and the individual message handlers. With message crackers, the function of the window procedure is principally to route each message to the appropriate handler.
Both the WM_LBUTTONDOWN and WM_LBUTTONDBLCLK messages map to the MyWnd_OnLButtonDown procedure. This mapping is one of the special cases of message handling described in Handling Special Cases of Messages.

```
// MYAPP.C
--------------------------------------------------------------------------

// MyWnd window procedure implementation.
//
LONG WINAPI MyWnd_WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM Param)
{
    switch (msg)
    {
        HANDLE_MSG(hwnd, WM_MOUSEMOVE,MyWnd_OnMouseMove);
        HANDLE_MSG(hwnd, WM_LBUTTONDOWN, MyWnd_OnLButtonDown);
        HANDLE_MSG(hwnd, WM_LBUTTONDBLCLK, MyWnd_OnLButtonDown);
        HANDLE_MSG(hwnd, WM_LBUTTONUP, MyWnd_OnLButtonUp);
    default:
        return MyWnd_DefProc(hwnd, msg, wParam, lParam);
    }
}

// Message handler function implementations:
//
void MyWnd_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags)
{
    .
```

```
      .
      .
      return FORWARD_WM_MOUSEMOVE( hwnd, x, y, keyFlags, MyWnd_DefProc);
}

void MyWnd_OnLButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y, UINT
keyFlags)
{
     .
      .
      .
      return FORWARD_WM_LBUTTONDOWN( hwnd, fDoubleClick, x, y, keyFlags,
MyWnd_DefProc);
}

void MyWnd_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags)
{
     .
      .
      .
      return FORWARD_WM_LBUTTONUP( hwnd, x, y, keyFlags, MyWnd_DefProc);
}
```

Note that the symbol MyWnd_DefProc is defined to represent **DefWindowProc**. The purpose of this definition is to make code more reusable. This approach assumes you have a similar definition in each application. For example, in an MDI child control procedure, you would have this definition:

```
#define MyWnd_DefProc     DefMDIChildProc
```

If you then copied your message handler to the MDI procedure, you would only need to change the prefix in MyWnd_DefProc to make the code you copied work correctly. Conversely, if your code used the explicit call to **DefWindowProc**, it could create a bug that would be difficult to track down when copied to the MDI code.

Handling Special Cases of Messages

As a general rule, there is one set of message crackers for each message: a message decoder and a message forwarder. Another rule is that each message handler you write should return the same value that your code would normally return for that message. The following messages present exceptions to these rules:

Message handler	Comment
OnCreate, OnNCCreate	**BOOL** return type: return TRUE if there are no errors. If FALSE is returned, a window will not be created.
OnKey	Handles both key up and key down messages. The extra parameter *fDown* indicates whether the key is down or up.
OnLButtonDown, OnRButtonDown	Handles both click (button down) and double-click messages. The extra parameter *fDoubleClick* indicates whether the message received is a double-click message.
OnChar	This handler is passed only by character, and not the virtual key or key flags information.

Writing Message Crackers for User-Defined Messages

You can use message crackers with window messages that you define, but you must write your own macros. The easiest way to do this is to copy and modify existing macros from WINDOWSX.H.
To understand how to write these macros, consider some of the message crackers defined in WINDOWSX.H:

```
/* BOOL Cls_OnCreate(HWND hwnd, CREATESTRUCT FAR* lpCreateStruct) */

#define HANDLE_WM_CREATE(hwnd, wParam, lParam, fn) \
    ((fn)(hwnd, (CREATESTRUCT FAR*)lParam) ? 0L : (LRESULT)-1L)

#define FORWARD_WM_CREATE(hwnd, lpCreateStruct, fn) \
    (BOOL)(DWORD)(fn)(hwnd, WM_CREATE, 0, (LPARAM)lpCreateStruct)
```

The message decoder (**HANDLE**_*msg*) should be defined as a function call, (fn), followed by *hwnd* and other parameters derived from *wParam* and *lParam*. The message forwarder (**FORWARD**_*msg*) performs the reverse operation on the parameters, putting information back together to restore *wParam* and *lParam* before making the function call (fn). Each of these macros must cast the return value so that the correct type is returned.
When calling the message crackers you write, be careful about variable message values. If your message value is a constant (such as WM_USER+100), you can use HANDLE_MSG with the message in a **switch** statement. However, if the message is registered with **RegisterWindowMessage**, it assigns a number at run time. In this situation, you can't use HANDLE_MSG, because variables cannot be used as **case** values. You must handle the message separately, in an **if** statement:

```
// In MyWnd class initialization code:
//
UINT WM_NEWMESSAGE= 0;

WM_NEWMESSAGE= RegisterWindowMessage("WM_NEWMESSAGE");
.
.
.
// In MyWnd_WndProc(): window procedure:
//
LONG WINAPI MyWnd_WndProc(HWND hwnd, WORD msg, WPARAM wParam,
                          LPARAM lParam)
{
    if (msg == WM_NEWMESSAGE)
        HANDLE_WM_NEWMESSAGE(hwnd, wParam, lParam, MyWnd_OnNewMessage);

    switch (msg)
    {
        HANDLE_MSG(hwnd, WM_MOUSEMOVE, MyWnd_OnMouseMove);
      .
      .
      .
}
```

Adapting Message Crackers to Special Cases

Message crackers can generally be used with all types of application code. However, certain situations require modifications in coding style.
Dialog Procedures, Window Subclassing, and Window Instance Data show how to adapt message-cracker coding techniques for dialog procedures, window subclassing, and window instance data.

Dialog Procedures

Dialog procedures return a **BOOL** value to indicate whether the message was processed. (Window procedures, in contrast, return a **LONG** value rather than a **BOOL**.) Therefore, to adapt a message cracker to dialog-procedure code, you must call the message handler and cast the value to **BOOL**.
Because you have to insert the **(BOOL)** cast, you can't use HANDLE_MSG. You must invoke the message-decoder macro explicitly. Here's an example that shows how you'd use message crackers in a dialog procedure:

```
BOOL MyDlg_OnInitDialog(HWND hwndDlg, HWND hwndFocus, LPARAM lParam);
void MyDlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);

BOOL WINAPI MyDlg_DlgProc(HWND hwndDlg, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    switch (msg)
    {
    //
    // Since HANDLE_WM_INITDIALOG returns an LRESULT,
    // we must cast it to a BOOL before returning.
    //
    case WM_INITDIALOG:
        return (BOOL)HANDLE_WM_INITDIALOG(hwndDlg, wParam, lParam,
MyDlg_OnInitDialog);

    case WM_COMMAND:
        HANDLE_WM_COMMAND(hwndDlg, wParam, lParam, MyDlg_OnCommand);
        return TRUE;
        break;

    default:
        return FALSE;
    }
}
```

Window Subclassing

When you use message crackers with a subclassed window procedure, the strategy described earlier for using message forwarders does not work. Recall that this strategy involves the following macro call:
return FORWARD_*msg***(** *parmlist***,** *defaultMsgProc* **);**
This use of a message forwarder (**FORWARD_***msg*) calls *defaultMsgProc* directly. But in a subclassed window procedure, you must call the window procedure of the superclass by using the API function **CallWindowProc**. The problem is that **FORWARD_***msg* calls the *defaultMsgProc* with four parameters, but **CallWindowProc** needs five parameters.
The solution is to write an intermediate procedure. For example, the intermediate procedure could be named test_DefProc:

```
FORWARD_WM_CHAR(hwnd, ch, cRepeat, test_DefProc);
```

The test_DefProc function calls **CallWindowProc** and prepends the address of the superclass function (in this case, test_lpfnwpDefProc) to the parameter list:

```
LONG test_DefProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
     return CallWindowProc(test_lpfnwpDefProc, hwnd, msg, wParam, lParam);
}
```

You need to write just one such procedure for each subclassed window in your application. Each time you use a message forwarder, you give this intermediate procedure as the function address instead of **DefWindowProc**. The following example code shows the complete context:

```
// Global variable that holds the previous window proc address of
// the subclassed window:
//
WNDPROC test_lpfnwpDefProc = NULL;

// Code fragment to subclass a window and store previous wndproc value:
//
void Subclasstest(HWND hwndtest)
{
    extern HINSTANCE g_hinsttest;    // Global application instance handle

    // SubclassWindow() is a macro API that calls SetWindowLong()
    // as appropriate to change the window proc of hwndtest.
    //
    test_lpfnwpDefProc = SubclassWindow(hwndtest,
            (WNDPROC)MakeProcInstance( (FARPROC)test_WndProc, g_hinsttest));
     .
     .
     .}

// Default message handler function
//
// This function invokes the superclasses' window procedure.  It
// must be declared with the same signature as any window proc,
// so it can be used with the FORWARD_WM_* macros.
//
LONG test_DefProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    return CallWindowProc(test_lpfnwpDefProc, hwnd, msg, wParam, lParam);
}
```

```
// test window procedure.  Everything here is the same as in the
// normal non-subclassed case: the differences are encapsulated in
// test_DefProc.
//
LONG WINAPI test_WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
        HANDLE_MSG(hwnd, WM_CHAR, test_OnChar);
        .
           .
           .
    default:
        //
        // Be sure to call test_DefProc(), NOT DefWindowProc()!
        //
        return test_DefProc(hwnd, msg, wParam, lParam);
    }
}

// Message handlers
//
void test_OnChar(HWND hwnd, UINT ch, int cRepeat)
{
    if (ch == testvalue)
    {
        // handle it here
    }
    else
    {
        // Forward the message on to test_DefProc
        //
        FORWARD_WM_CHAR(hwnd, ch, cRepeat, test_DefProc);
    }
}
```

Window Instance Data

It is common for a window to have user-declared state variables (or "instance data") kept in a separate data structure allocated by the application. You associate this data structure with its corresponding window by storing a pointer to the structure in a specially named window property or in a window word (allocated by setting the cbWndExtra field of the **WNDCLASS** structure when the class is registered).
Message crackers can be adapted to work with this use of instance data. Place the *hwnd* of the window in the first member of the structure. Then, in the message decoders (**HANDLE**_*msg* macros), pass the address of the structure instead of the *hwnd*. The message handler now gets a pointer to the structure instead of the *hwnd*, but it can access the *hwnd* through indirection. You may need to rewrite some of the message handler to make it use indirection to access the window handle.
The following example illustrates this technique:

```
// Window instance data structure.  Must include window handle field.
//
typedef struct _test
{
    HWND hwnd;
    int otherStuff;
} test;

// "test" window class was registered with cbWndExtra = sizeof(test*), so we
// can use a window word to store back pointer.  Window properties can also
// be used.
//
// These macros get and set the pointer to the instance data corresponding
to the
// window.  Use GetWindowWord or GetWindowLong as appropriate based on the
default
// size of data pointers.
//
#ifdef WIN32
#define test_GetPtr(hwnd)        (test*)GetWindowLong((hwnd), 0)
#define test_SetPtr(hwnd, ptest)  (test*)SetWindowLong((hwnd), 0, (LONG)
(ptest))
#else
#define test_GetPtr(hwnd)        (test*)GetWindowWord((hwnd), 0)
#define test_SetPtr(hwnd, ptest)  (test*)SetWindowWord((hwnd), 0, (WORD)
(ptest))
#endif

// Default message handler

#define test_DefProc DefWindowProc

// Message handler functions, declared with a test* as their first argument,
// rather than an HWND.  Other than that, their signature is identical to
// that shown in WINDOWSX.H.
//
BOOL test_OnCreate(test* ptest, CREATESTRUCT FAR* lpcs);
void test_OnPaint(test* ptest);
```

```
//
// Code to register the test window class:
//
BOOL test_Init(HINSTANCE hinst)
{
    WNDCLASS cls;

    cls.hCursor         = ...;
    cls.hIcon           = ...;
    cls.lpszMenuName    = ...;
    cls.hInstance       = hinst;
    cls.lpszClassName   = "test";
    cls.hbrBackground   = ...;
    cls.lpfnWndProc     = test_WndProc;
    cls.style           = CS_DBLCLKS;
    cls.cbWndExtra      = sizeof(test*);  // room for instance data ptr
    cls.cbClsExtra      = 0;

    return RegisterClass(&cls);
}

// The window proc for class "test".  This demonstrates how instance data is
// attached to a window and passed to the message handler functions.
//
LRESULT CALLBACK test_WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    test* ptest = test_GetPtr(hwnd);

    if (ptest == NULL)
    {
    // If we're creating the window, then try to allocate it.
    //
     if (msg == WM_NCCREATE)
     {
           // Create the instance data structure, set up the hwnd
backpointer
           // field, and associate it with the window.
           //
           ptest = (test*)LocalAlloc(LMEM_FIXED | LMEM_ZEROINIT,
sizeof(test));

           // If an error occured, return 0L to fail the CreateWindow call.
           // This will cause CreateWindow() to return NULL.
           //
           if (ptest == NULL)
           return 0L;
```

```
            ptest->hwnd = hwnd;
            test_SetPtr(hwnd, ptest);

            // NOTE: the rest of the test structure should be initialized
            // inside of Template_OnCreate() (or Template_OnNCCreate()).
Further
            // creation data may be accessed through the CREATESTRUCT FAR*
parameter.
            //
     }
     else
     {
            // It turns out WM_NCCREATE is NOT necessarily the first message
            // received by a top-level window (WM_GETMINMAXINFO is).
            // Pass messages that precede WM_NCCREATE on through to
            // test_DefProc
            //
            return test_DefProc(hwnd, msg, wParam, lParam);
     }
    }

    if (msg == WM_NCDESTROY)
    {
     LocalFree((HLOCAL)ptest);

            ptest = NULL;
     test_SetPtr(hwnd, NULL);

    }

    switch (msg)
    {
    HANDLE_MSG(ptest, WM_CREATE, test_OnCreate);
    HANDLE_MSG(ptest, WM_PAINT, test_OnPaint);
    ...

    default:
     return test_DefProc(hwnd, msg, wParam, lParam);
    }
}
```

Using Control Message Functions

The control message API functions perform a role that is the converse of message crackers: instead of handling messages sent to your window, they send messages to other windows (controls).
Each of the control message functions packs parameters into the appropriate *wParam*/*lParam* format and then calls **SendMessage**. These functions offer the same portability advantages that message crackers do; they free you from having to know how the current operating system packs *wParam* and *lParam*.
The function calls also improve readability of code and support better type checking. When used with the STRICT enhancements, the control message functions help prevent incorrect passing of message parameters.
To see how the control message functions work, first look at the following source code, which makes two calls to **SendMessage** to print all the lines in an edit control:

```
void PrintLines(HWND hwndEdit, WHND hwndDisplay)
{
    int line;
    int lineLast = (int)SendMessage(hwndEdit, EM_GETLINECOUNT, 0, 0L);

    for (line = 0; line < lineLast; line++)
    {
        int cch;
        char ach[80];

        *((LPINT)ach) = sizeof(ach);
        cch = (int)SendMessage(hwndEdit, EM_GETLINE,
                line, (LONG)(LPSTR)ach);

        PrintInWindow(ach, hwndDisplay);
    }
}
```

The source code below uses two control message functions, **Edit_GetLineCount** and **Edit_GetLine**, to perform the same task. This version of the code is shorter, easier to read, doesn't generate compiler warnings, and doesn't have any non-portable casts:

```
void PrintLines(HWND hwndEdit, WHND hwndDisplay)
{
    int line;
    int lineLast = Edit_GetLineCount(hwndEdit);

    for (line = 0; line < lineLast; line++)
    {
        int cch;
        char ach[80];

        cch = Edit_GetLine(hwndEdit, line, ach, sizeof(ach));

        PrintInWindow(ach, hwndDisplay);
    }
}
```

The control message API functions are listed in the table below. For more information, refer to the macro definitions in WINDOWSX.H and the documentation for the corresponding window message.

Control Message API Functions

Control Group	Functions
Static Text Controls:	**Static_Enable**(*hwnd*, *fEnable*)

Static_GetIcon(*hwnd*, *hIcon*)
Static_GetText(*hwnd*, *lpch*, *cchMax*)
Static_GetTextLength(*hwnd*)
Static_SetIcon(*hwnd*, *hIcon*)
Static_SetText(*hwnd*, *lpsz*)

Button Controls:

Button_Enable(*hwnd*, *fEnable*)
Button_GetCheck(*hwnd*)
Button_GetState(*hwnd*)
Button_GetText(*hwnd*, *lpch*, *cchMax*)
Button_GetTextLength(*hwnd*)
Button_SetCheck(*hwnd*, *check*)
Button_SetState(*hwnd*, *state*)
Button_SetStyle(*hwnd*, *style*, *fRedraw*)
Button_SetText(*hwnd*, *lpsz*)

Edit Controls:

Edit_CanUndo(*hwnd*)
Edit_EmptyUndoBuffer(*hwnd*)
Edit_Enable(*hwnd*, *fEnable*)
Edit_FmtLines(*hwnd*, *fAddEOL*)
Edit_GetFirstVisible(*hwnd*)
Edit_GetHandle(*hwnd*)
Edit_GetLine(*hwnd*, *line*, *lpch*, *cchMax*)
Edit_GetLineCount(*hwnd*)
Edit_GetModify(*hwnd*)
Edit_GetRect(*hwnd*, *lprc*)
Edit_GetSel(*hwnd*)
Edit_GetText(*hwnd*, *lpch*, *cchMax*)
Edit_GetTextLength(*hwnd*)
Edit_LimitText(*hwnd*, *cchMax*)
Edit_LineFromChar(*hwnd*, *ich*)
Edit_LineIndex(*hwnd*, *line*)
Edit_LineLength(*hwnd*, *line*)
Edit_ReplaceSel(*hwnd*, *lpszReplace*)
Edit_Scroll(*hwnd*, *dv*, *dh*)
Edit_SetHandle(*hwnd*, *h*)
Edit_SetModify(*hwnd*, *fModified*)
Edit_SetPasswordChar(*hwnd*, *ch*)
Edit_SetRect(*hwnd*, *lprc*)
Edit_SetRectNoPaint(*hwnd*, *lprc*)
Edit_SetSel(*hwnd*, *ichStart*, *ichEnd*)
Edit_SetTabStops(*hwnd*, *cTabs*, *lpTabs*)
Edit_SetText(*hwnd*, *lpsz*)
Edit_SetWordBreak(*hwnd*, *lpfnWordBreak*)
Edit_Undo(*hwnd*)

Scroll Bar Controls:

ScrollBar_Enable(*hwnd*, *flags*)
ScrollBar_GetPos(*hwnd*)
ScrollBar_GetRange(*hwnd*, *lpposMin*, *lpposMax*)
ScrollBar_SetPos(*hwnd*, *pos*, *fRedraw*)
ScrollBar_SetRange(*hwnd*, *posMin*, *posMax*, *fRedraw*)
ScrollBar_Show(*hwnd*, *fShow*)

List Box Controls:

ListBox_AddFile(*hwnd*, *lpszFilename*)
ListBox_AddItemData(*hwnd*, *data*)
ListBox_AddString(*hwnd*, *lpsz*)
ListBox_DeleteString(*hwnd*, *index*)
ListBox_Dir(*hwnd*, *attrs*, *lpszFileSpec*)
ListBox_Enable(*hwnd*, *fEnable*)
ListBox_FindItemData(*hwnd*, *indexStart*, *data*)
ListBox_FindString(*hwnd*, *indexStart*, *lpszFind*)
ListBox_GetAnchorIndex(*hwnd*)
ListBox_GetCaretIndex(*hwnd*)
ListBox_GetCount(*hwnd*)
ListBox_GetCurSel(*hwnd*)
ListBox_GetHorizontalExtent(*hwnd*)
ListBox_GetItemData(*hwnd*, *index*)
ListBox_GetItemHeight(*hwnd*, *index*) (1)
ListBox_GetItemRect(*hwnd*, *index*, *lprc*)
ListBox_GetSel(*hwnd*, *index*)
ListBox_GetSelCount(*hwnd*)
ListBox_GetSelItems(*hwnd*, *cItems*, *lpIndices*)
ListBox_GetText(*hwnd*, *index*, *lpszBuffer*)
ListBox_GetTextLen(*hwnd*, *index*)
ListBox_GetTopIndex(*hwnd*)
ListBox_InsertItemData(*hwnd*, *lpsz*, *index*)
ListBox_InsertString(*hwnd*, *lpsz*, *index*)
ListBox_ResetContent(*hwnd*)
ListBox_SelectItemData(*hwnd*, *indexStart*, *data*)
ListBox_SelectString(*hwnd*, *indexStart*, *lpszFind*)
ListBox_SelItemRange(*hwnd*, *fSelect*, *first*, *last*)
ListBox_SetAnchorIndex(*hwnd*, *index*)
ListBox_SetCaretIndex(*hwnd*, *index*)
ListBox_SetColumnWidth(*hwnd*, *cxColumn*)
ListBox_SetCurSel(*hwnd*, *index*)
ListBox_SetHorizontalExtent(*hwnd*, *cxExtent*)
ListBox_SetItemData(*hwnd*, *index*, *data*)
ListBox_SetItemHeight(*hwnd*, *index*, *cy*) (1)
ListBox_SetSel(*hwnd*, *fSelect*, *index*)
ListBox_SetTabStops(*hwnd*, *cTabs*, *lpTabs*)

ListBox_SetTopIndex(*hwnd*, *indexTop*)

Combo Box Controls:

ComboBox_AddItemData(*hwnd*, *data*)
ComboBox_AddString(*hwnd*, *lpsz*)
ComboBox_DeleteString(*hwnd*, *index*)
ComboBox_Dir(*hwnd*, *attrs*, *lpszFileSpec*)
ComboBox_Enable(*hwnd*, *fEnable*)
ComboBox_FindItemData(*hwnd*, *indexStart*, *data*)
ComboBox_FindString(*hwnd*, *indexStart*, *lpszFind*)
ComboBox_GetCount(*hwnd*)
ComboBox_GetCurSel(*hwnd*)
ComboBox_GetDroppedControlRect(*hwnd*, *lprc*) (1)
ComboBox_GetDroppedState(*hwnd*) (1)
ComboBox_GetEditSel(*hwnd*)
ComboBox_GetExtendedUI(*hwnd*) (1)
ComboBox_GetItemData(*hwnd*, *index*)
ComboBox_GetItemHeight(*hwnd*)
ComboBox_GetLBText(*hwnd*, *index*, *lpszBuffer*)
ComboBox_GetLBTextLen(*hwnd*, *index*)
ComboBox_GetText(*hwnd*, *lpch*, *cchMax*)
ComboBox_GetTextLength(*hwnd*)
ComboBox_InsertItemData(*hwnd*, *index*, *data*)
ComboBox_InsertString(*hwnd*, *index*, *lpsz*)
ComboBox_LimitText(*hwnd*, *cchLimit*)
ComboBox_ResetContent(*hwnd*)
ComboBox_SelectItemData(*hwnd*, *indexStart*, *data*)
ComboBox_SelectString(*hwnd*, *indexStart*, *lpszSelect*)
ComboBox_SetCurSel(*hwnd*, *index*)
ComboBox_SetEditSel(*hwnd*, *ichStart*, *ichEnd*)
ComboBox_SetExtendedUI(*hwnd*, *flags*) (1)
ComboBox_SetItemData(*hwnd*, *index*, *data*)
ComboBox_SetItemHeight(*hwnd*, *cyItem*) (1)
ComboBox_SetText(*hwnd*, *lpsz*)
ComboBox_ShowDropdown(*hwnd*, *fShow*)

1 Supported only for Win32, not for Windows 3.*x*. These functions are not available if you define the symbol WINVER as equal to 0x0300, on the command line or with a **#define** statement.

Improving Application Performance on Windows NT

This overview provides some guidelines and hints to help you improve application performance on Windows NT. The sophisticated virtual memory manager in Windows NT permits applications to have direct access to very large data structures. Increased protection permits applications to be less concerned about cooperating with other applications, and more focused on being responsive to the user. However, there are costs associated with increased protection and portability to multiple processors that require some changes to the coding styles you used with earlier versions of Windows.

Both Windows NT and Windows 95 implement the Win32 API. If your application must run on both platforms, you must be careful that design decisions that improve performance on one platform do not seriously degrade performance on the other platform.

Managing Kernel Resources

The following sections discuss using kernel resources effectively to improve performance.

Managing Memory

Windows NT can address 4 gigabytes of memory. Each application running on Windows NT has most of the lower 2 gigabytes of its virtual address space at its disposal. For a console application, the system uses 5.5 MB of the lower 2 gigabytes to permit you to view portions of the system that reside elsewhere. For a Windows application, the system uses to 9 MB for that purpose.
If your application is being ported from another operating system or from an earlier version of Windows, you might have developed a special virtual memory scheme for your own private use. Having another memory manager for your application decreases performance. To increase performance, let Windows NT manage the virtual address space.
Consider using file mapping in the following situations.

- You are randomly accessing a read-only file or a file that is only written to by one process. Shared writing to memory-mapped files from multiple processes requires a bit of internal system synchronization and does not work well if the file is remote, because you will have to manage the remote synchronization. Using memory-mapped files for sequential file access is faster than standard sequential file access, but uses more memory than the file system cache does. If you are going to access a file sequentially, use the FILE_FLAG_SEQUENTIAL_SCAN flag when calling the **CreateFile** function.
- You are using a temporary file and you know its maximum size. You can map a large temporary space which is backed by the system paging files instead of by an existing file. Simply pass 0xffffffff as the file handle to the **CreateFileMapping** function and specify the size you need. You can also call the **VirtualAlloc** function to create a large data space backed by the paging file, but this memory is not sharable with another process. However, do not create a large area backed by disk if your application must run on machines with limited disk space. Instead, use **VirtualAlloc** to reserve that amount of linear address space, then commit pages as you need them.
- You are managing communication between multiple processes. Memory-mapped files are faster than other mechanisms, such as named pipes, RPC, or shared file access. You might need a mutex to protect access to the shared section, but mutex operations are inexpensive.

Managing Files

- If you are processing a file sequentially, use a page at a time to reduce the number of calls you have to make to the file system. If you are randomly accessing small amounts of data, you may wish to use memory-mapped files.
- In MS-DOS-based systems, there is a limit on the number of files the system can have open at one time. Therefore, many applications open and close files frequently. Because of the additional protection and security in Windows NT, opening a file uses more resources. Therefore, on Windows NT, open files when you first need them, and close them when you are finished with them. The number of files you can open is limited only by the amount of non-paged pool in the system. The system stores information for all open objects in non-paged pool.

Managing Processes

- Using multiple threads to improve performance, especially from server applications or computers with multiple processors. You can also use multiple threads to allow your application to be more responsive to user input. As an alternative, Windows NT supports asynchronous file access. You can issue file requests and have the system notify you when they have completed. This is more efficient than having a separate thread for each outstanding concurrent file request.
- Use the **SetPriorityClass** function to allow your process to use the real-time priority class. This is useful for an application which is processing data in real time or doing time-sensitive communication with an external device. Your application should not use the real-time priority class for very long or you it will preempt all activity on the system, including the work of system processes.
- Call the **VirtualLock** function to identify a small number of pages to retain in memory, so you do not have to wait for normal paging when attempting to respond to a real-time device. Your design should minimizes the amount of code that executes in the real-time priority class with locked pages. You can use event objects and memory-mapped files to exchange information with processes running at normal priority.
- When storing and retrieving data from the registry, use the data type MULTI_SZ. This data type allows you to store a set of data values under the name of a single value by concatenating the strings into a single multistring. A multistring has multiple individual strings separated by a /0 character, with the last one followed by an additional /0 character. One registry call retrieves all the strings. This is very efficient, especially if the value is accessed remotely.
- Windows NT uses Unicode internally. Unicode is a 16-bit character-coding standard which includes symbols for all international languages. From a performance viewpoint, it is better to write the application to work with Unicode. This decreases overhead and make the application easier to port to foreign languages.

Managing Graphics

Whenever possible be sure to use **PolyTextOut**, **PolyPolyline**, **PolylineTo**, **PolyDraw**, **PolyBezier**, and **PolyBezierTo** functions. These functions exploit the fact that many drawing calls use identical attributes, and so multiple items can be drawn in a single call once the brushes, pens, colors, and fonts have been selected. For example, the console window uses **PolyTextOut**. This change reduced scrolling time in a console window by 30% when it was implemented during the development of Windows NT.

If you are writing an application that draws on the display, then the **CreateDIBSection** function can improve performance. This function allows you to share a memory section directly with the system, and thus avoid having it copied from your process to the system each time there is a change. Previously, a common practice was to call the **GetDIBits** function, make the required changes, then call the **SetDIBits** function. These steps were often repeated on different scan lines of the bitmap before the image was ready for updating. Using **CreateDIBSection** is much simpler.

One word of caution if you decide to use **CreateDIBSection**. You need to be sure that any calls that might affect your bitmap have completed before you start to draw in it. This is because the batching of GDI calls may cause their delayed execution. For example, suppose you make a **PatBlt** call to clear your bitmap, then you start to change the bits in your DIB section. If the **PatBlt** call was batched, it might not actually execute until after you start to make the bitmap changes. So, before you make changes to your DIB section, be sure to call **GdiFlush** if you have made changes to the bitmap with earlier GDI calls.

Batch Processing for Graphics

GDI calls are handled using batch processing. Each thread has a batch limit. The GDI calls for a thread are not sent until the batch limit is reached or until the thread calls a function which flushes the batch. To increase performance, make sure that the batch limit is as large as appropriate and that your code is structured to minimize flushing the batch.

Three function calls help you manage the batching of GDI calls. The **GdiSetBatchLimit** function allows you to raise and lower the batch limit, which defaults to ten. For best performance, you should set the limit as high as possible while avoiding jerky drawing on the display. You will want to test any changes to the batch limit on a very slow machine and a very fast one to be sure you have not introduced a problem which will only appear in one environment or the other. You can call **GdiGetBatchLimit** to determine the current limit. Finally, you can call **GdiFlush** to flush the batch at the end of an operation you would like to see displayed immediately.

In general, you can batch graphical output functions that return a Boolean value, indicating success or failure. A few frequently used functions that return non-Boolean results which were seldom used have new replacement calls that just return Boolean results. The **SetPixelV** and **MoveToEx** functions are two important cases.

Most calls that manipulate the window system flush the batch. One reason is that much of the window system is visible to all processes on the desktop and so the central data repository. The **PeekMessage** function does not flush the batch, but the **GetMessage** function does. Graphics calls that return a handle or a number flush the batch. An important exception to this is the group of calls for selecting fonts, brushes, and pens. These calls are batched. However, selecting bitmaps and regions flush the batch. So do the **SetWorldTransform** and **SetMapMode** functions. When possible, you should organize your code so that GDI calls that do not flush the batch are grouped together, then make the calls that flush the batch.

There are circumstances in which you will want to minimize the batching of GDI calls. For example, when you need the display to immediately reflect the drawing you do, you want to flush the batch explicitly no matter how large or small it is. Failing to do this causes the data to be updated with noticeably odd timing. Also, you want to minimize the batch when you are debugging your application. Otherwise, an error returned in the middle of a batch may not be received until some other call flushes the batch; it will appear then that the wrong call failed. Finally, if you are obtaining certain performance measurements on your application, you will want to set the batch to one.

Managing the Device Context

In 16-bit Windows, it was important to conserve the use of drawing objects. In Windows NT, there are richer data structures to hold a new wealth of data. That can mean it takes longer to look things up.
The old limitations gave rise to a coding style in which you created, selected, used, and destroyed objects (like pens and brushes) constantly. This limited the number of objects in the system and kept the application from running into the limits of the address space. Because of the transition to a client-server architecture in Windows NT, object creation and destruction are more expensive. Because of the new capacity for large numbers of objects, selecting objects is also a bit slower. For example, the old coding style

```
select(grey);
patblt(...);
select(black);
patblt(...);
select(grey);
patblt(...);
select(black);
patblt(...);
```

is slower on Windows NT than

```
select(grey);
patblt(...);
patblt(...);
patblt(...);
select(black);
patblt(...);
patblt(...);
patblt(...);
```

In Windows NT, create all your objects when you first need them. Do not destroy the objects until you are done with them.
If you were programming for 16-bit Windows, you were told to avoid the use of your own DC's because the system could only support a few. This is not true on Windows NT. Use the creation style CS_OWNDC as much as you can when you call the **RegisterClass** function. This avoids repeated calling of the relatively expensive **GetDC** and **ReleaseDC** functions every time you have to draw. It also preserves the selected objects in your own DC in between calls, eliminating the need to select them again after each call to **GetDC**.

Managing Asynchronous Input

One of the biggest difference between 16-bit Windows and Windows NT is the input model. On 16-bit Windows, there is an synchronous input model, sometimes called cooperative multitasking. In this model, each application must process its messages, otherwise, it would hold up all the other applications on the system. A common coding style involved frequently calling the **PeekMessage** function, because every application constantly had to check the message queue for messages and pass them on. If they did not, the system would appear to hang.
On Windows NT, there is an asynchronous input model. In this model, messages are sent only to the processes that need to see them. If one process ceases to deal with its messages, it may become unresponsive and cease to update its display area, but the rest of the system is not affected. This means that **PeekMessage** need only be called to remain responsive to the user during a lengthy operation.

Considerations for RISC Computers

Once you write an application for the x86 version of Windows NT, it is easy it is to get it to run on RISC versions of Windows NT. This is because there are virtually no processor dependencies in the Win32 API.
However, an application's performance can suffer on a RISC machine if its data is not properly aligned. To handle this problem, align the data in your source for both RISC and non-RISC machines. You want to assure that you have DWORDs on DWORD boundaries, and LARGE_INTEGERS on 8-byte boundaries.
Normally the compiler makes sure that data is correctly aligned. However, data structures from a file or from over a network may not follow alignment rules. With the Microsoft Visual C++ compiler, you will want to use the PACK and UNPACK pragmas to define these structures, and the modifier UNALIGNED to declare pointers to them. If you do not do this, you get alignment faults. On some systems, these simply generate traps, and you can fix your program. However, some systems handle your unaligned references with a trap handler. This slows down your application and can be difficult to track down.

Overview of the Build Process

This guide describes how to build 32-bit applications for Windows and dynamic-link libraries (DLLs) using the Microsoft Win32 Software Development Kit (SDK) and your development tools. It describes the components of a generalized makefile and includes information on using the C run-time libraries.
The following table summarizes the steps used to build a Win32-based application or DLL:

Step	Tool
Compile C and C++ source-language files into object files.	C/C++ compiler included with your development tools.
Create and edit resources. Doing so may also create include (.H) files which define useful constants.	Dialog Editor, Image Editor, and Font Editor included with the SDK or equivalent development tools.
Compile resource scripts into linkable resource files.	Resource Compiler included with the SDK or your development tools.
Create import libraries for each DLL.	Library manager included with your development tools.
Link the object modules, resources, standard libraries, and import libraries (for an executable using DLLs) to produce an executable module.	Linker included with your development tools.
Tune your application.	Performance tools and Working Set Tuner included with the SDK.

The sample makefiles provided with the SDK samples give good examples of the build process. Each of them includes WIN32.MAK, which defines most of the common macros you need to build 32-bit applications for Windows NT and Windows 95 using Microsoft development tools.
The remainder of this guide is specific to Microsoft development tools.

Using WIN32.MAK

The WIN32.MAK file defines macros that can be used to simplify your own makefiles for use with Microsoft development tools. These macros help assure that you have chosen the correct options for the following tasks:

- Setting Targets
- Compiling Source Files
- Building Applications
- Building DLLs
- Using the C Run-Time Library

The file should be included in your makefile as follows:

```
!include <win32.mak>
```

It is recommended that you examine the contents of WIN32.MAK, located in the MSTOOLS\INCLUDE subdirectory.

Setting Targets

WIN32.MAK includes the following macros that control the type of target that is built. Set these macros in your makefile prior to including WIN32.MAK or set them as environment variables.

- **APPVER**

  Specifies the version of the application. The value includes major and minor versions, (i.e. 3.51 or 4.0). It is used for checking version dependencies and for marking the executable with version information.
- **TARGETOS**

  Specifies the operating system. The value can be WIN95, WINNT, or BOTH. It is used for checking platform dependencies.
- **TARGETLANG**

  Specifies the language. The value can be Japanese. The default value is the system locale.

In addition, you may set one of these values to 1 on the NMAKE command line or as an environment variable.

- **NODEBUG**

  Specifies no debugging information should be included in the output files.
- **PROFILE**

  Specifies that information for the profiler should be included in the output files.
- **TUNE**

  Specifies that information for the working set tuner (WST) should be included in the output files.

For example, the following command line specifies that no debugging information should be included in the output files:

```
nmake NODEBUG=1
```

Compiling Source Files

WIN32.MAK includes the following macros to simplify compilation of C/C++ source files:

- **$(CC)**

  This macro is used to invoke the compiler. It expands to the following:

  ```
  cl
  ```

- **$(CFLAGS)**

  This macro is used for common compiler flags. It is platform dependent. On x86 machines, it includes the following flags:

  ```
  -c -W3 -D_X86=1
  ```

- **$(CDEBUG)**

  This macro is used for debugging, profiling, and tuning compiler flags. When specifying debug, but no profiling or tuning, it expands to the following flags:

  ```
  -Z7 -Od
  ```

Use one of the following three macros if you are using the C run-time library:

- **$(CVARS)**

  This macro is used for single-threaded applications. It expands to the following flags:

  ```
  -DWIN32 -DNULL=0
  ```

- **$(CVARSMT)**

  This macro is used for multithreaded applications. It expands to the following flags:

  ```
  -DWIN32 -DNULL=0 -D_MT
  ```

- **$(CVARSDLL)**

  This macro is used for applications that use the CRT in a DLL. It expands to the following flags:

  ```
  -DWIN32 -DNULL=0 -D_MT -D_DLL
  ```

The following example shows how you can compile your source files using these macros:

```
generic.obj: generic.c
    $(cc) $(cdebug) $(cflags) $(cvarsdll) $*.c
```

WIN32.MAK includes the following macros to simplify compilation of resource-definition files:

- **$(RC)**

  This macro is used to invoke the resource compiler. It expands to the following:

  ```
  rc
  ```

- **$(RCFLAGS)**

  This macro is used for common flags. It expands to the following flags:

  ```
  -r
  ```

- **$(RCVARS)**

  This macro is used for other flags. It expands to the following flags:

  ```
  -DWIN32 -DWINVER=0x0400
  ```

The following example shows how you can compile your resource-definition file using these macros:

```
generic.res: generic.rc generic.h generic.dlg
    $(rc) $(rcvars) $(rcflags) $*.rc
```

Building Applications

WIN32.MAK contains the following macros to simplify linking applications:

- **$(LINK)**

  This macro is used to invoke the linker. It expands to the following flags:

  ```
  link
  ```

- **$(LDEBUG)**

  This macro is used for for specifying debugging, profiling, and tuning linker flags. When specifying debug, but no profiling or tuning, it expands to the following flags:

  ```
  -debug:full -debugtype:cv
  ```

Use one of the following flags, depending on whether you are building a console application or a GUI application:

- **$(CONLFLAGS)**

  This macro is used for console applications. It is platform dependent. On x86 machines, it expands to the following flags:

  ```
  -/NODEFAULTLIB /INCREMENTAL:NO /PDB:NONE /RELEASE /NOLOGO
  -align:0x1000 -subsystem:console,4.0 -entry:mainCRTStartup
  ```

- **$(GUIFLAGS)**

  This macro is used for GUI applications. it is platform dependent. On x86 machines, it expands to the following flags:

  ```
  -/NODEFAULTLIB /INCREMENTAL:NO /PDB:NONE /RELEASE /NOLOGO
  -align:0x1000 -subsystem:windows,4.0 -entry:WinMainCRTStartup
  ```

Notice that these macros provide entry points in the CRT libraries. These CRT entry points will call your entry-point function (main or WinMain).
Use one of the following flags if you are using the single-threaded CRT:

- **$(CONLIBS)**

  This macro is for console applications using LIBC.LIB. It expands to the following list of libraries:

  ```
  LIBC.LIB OLDNAMES.LIB KERNEL32.LIB ADVAPI32.LIB
  ```

- **$(GUILIBS)**

  This macro is used for GUI application using LIBC.LIB. It expands to the following list of libraries:

  ```
  LIBC.LIB OLENAMES.LIB KERNEL32.LIB ADVAPI32.LIB USER32.LIB GDI32.LIB
  COMDLG32.LIB WINSPOOL.LIB
  ```

Use one of the following flags if you are using the multithreaded CRT:

- **$(CONLIBSMT)**

  This macro is for console applications using LIBCMT.LIB. It expands to the following list of libraries:

  ```
  LIBCMT.LIB OLDNAMES.LIB KERNEL32.LIB ADVAPI32.LIB
  ```

- **$(GUILIBSMT)**

  This macro is used for GUI application using LIBCMT.LIB. It expands to the following list of libraries:

  ```
  LIBCMT.LIB OLDNAMES.LIB KERNEL32.LIB ADVAPI32.LIB USER32.LIB GDI32.LIB
  COMDLG32.LIB WINSPOOL.LIB
  ```

Use one of the following flags if you are using the CRT in a DLL:

- **$(CONLIBSDLL)**
  This macro is for console applications using MSVCRT.LIB. It expands to the following list of libraries:
  `MSVCRT.LIB OLDNAMES.LIB KERNEL32.LIB ADVAPI32.LIB`

- **$(GUILIBSDLL)**
  This macro is used for GUI application using MSVCRT.LIB. It expands to the following list of libraries:
  `MSVCRT.LIB OLDNAMES.LIB KERNEL32.LIB ADVAPI32.LIB USER32.LIB GDI32.LIB COMDLG32.LIB WINSPOOL.LIB`

The following example links the object files and libraries to produce the application executable file. Note that the resource file (generic.res) is linked along with the object file (generic.obj).

```
generic.exe : generic.obj generic.res
    $(link) $(ldebug) $(guilflags) generic.obj generic.res \
        $(guilibsdll) -out:generic.exe
```

Building DLLs

To build a DLL, use the macros defined for building applications. WIN32.MAK also includes the following macros to simplify linking DLLS:

- **$(IMPLIB)**

  This macro is used to invoke the library manager. It expands to the following flags:

  ```
  lib
  ```

- **$(DLLENTRY)**

  This macro is used as a suffix for the entry-point function. It is platform dependent. It defines to nothing on MIPS, Alpha, and PPC machines. It expands to the following flags on x86 machines:

  ```
  @12
  ```

- **$(DLLLFLAGS)**

  This macro is used for DLLs. It expands to the following flags:

  ```
  -/NODEFAULTLIB /INCREMENTAL:NO /PDB:NONE /RELEASE /NOLOGO
  -align:0x1000 -entry:_DllMainCRTStartup$(DLLENTRY) -dll
  ```

The following example illustrates how to create the import library for a DLL. The import library is linked with the application that uses this DLL.

```
mydll.lib: mydll.def mydll.obj
    $(implib) -machine:$(CPU) -def:mydll.def mydll.obj -out:mydll.lib
```

The following example links the DLL:

```
mydll.dll : mydll.obj
    $(link) $(ldebug) $(dlllflags) -base:0x1C000000 mydll.obj \
        $(conlibsdll) -out:mydll.dll
```

Using the C Run-Time Library

The following sections describe how to use the different forms of the Microsoft C Run-time Library (CRT) when building your application.
Microsoft Visual C++ contains three forms of the CRT:

LIBC.LIB
Statically linked library for single-threaded applications.

LIBCMT.LIB
Statically linked library that supports multi-threaded applications.

MSVCRT.LIB
Import library for the CRT in a DLL that also supports multi-threaded applications.

The CRT libraries are not distributed with the Win32 SDK. Many vendors distribute similar forms of the CRT, but with different names. These name differences can be encapsulated in WIN32.MAK.

Note If you would like to avoid linking in the CRT, compile your source files with the **-Qlfdiv-** flag, do not call the CRT functions, do not call any math routines, and do not specify an entry point in the CRT with the **-entry** linker option.

Calling the C Run-Time Library from a DLL

When linking a DLL with any of the C run-time libraries, you must specify the following the entry point for the DLL.

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

This entry point is exported from the C run-time libraries and performs necessary initialization. The **$(DLLFLAGS)** macro in WIN32.MAK includes the -entry flag shown above.
DllMainCRTStartup will call your entry point if you name it **DllMain** and export it from your DLL. Therefore, your DLL entry-point function will look something like this:

```
BOOL WINAPI DllMain( HINSTANCE hinstDLL,
                     DWORD fdwReason,
                     LPVOID lpReserved )
{

   if( fdwReason==DLL_PROCESS_ATTACH || fdwReason==DLL_THREAD_ATTACH )
   ...
   if( fdwReason==DLL_PROCESS_DETACH || fdwReason==DLL_THREAD_DETACH )
   ...

   return( TRUE );

}
```

For more information, see **DllEntryPoint**.

Mixing Library Types

If your DLL is linked with MSVCRT.LIB, any modules that call your DLL should also be linked with MSVCRT.LIB. Otherwise, unpredictable results can occur, because the module and the DLL have separate copies of all CRT functions and global variables. To avoid problems, link both the executable and the DLL with MSVCRT.LIB. This allows both the executable and the DLL to use the common set of functions and data contained in MSVCRT.DLL. C run-time data such as stream handles can then be shared by both the executable and the DLL. If you must mix library types, be sure to use the following guidelines:

- CRT file handles may only be used by the module that created them.
- FILE pointers may only be used by the module that created them.
- Memory allocated with the CRT may only be freed or reallocated by the module that allocated.

If a DLL is linked with LIBC.LIB, and the DLL may be called by a multi-threaded application, multiple threads running in this DLL at the same time will not be supported. This may cause unpredictable results. Therefore, if there is a possibility that the DLL will be called by multi-threaded programs, be sure to link it with one of the libraries that support multi-threaded programs (LIBCMT.LIB or MSVCRT.LIB).

Installing Applications

This article describes a standard set of guidelines for installing Win32-based applications.

About Installing Applications

The purpose of the guidelines described in this article is to enable all application developers to support the same general method of application installation. The prime benefit is for users, many of whom have said they prefer a consistent installation method so that they do not need to learn a different method with each new software purchase. These guidelines also benefit the application developer by helping to standardize the organization and management of application files, thereby making initial installations, updates, and application removals easier.

Installation Program

The installation program plays the primary role in carrying out application installation. The program retrieves information from the user and the computer and installs the files and information needed to run the application successfully. Every installation program carries out these basic steps:

1. Determines the user's hardware and software configuration and available disk space.
2. Copies application executable and data files to the appropriate directories on the hard disk.
3. Sets up the execution environment for the application by modifying existing files and adding entries to the registry.

An installation program (or a companion program) should also be prepared to update or remove an already installed application.
You are responsible for designing and implementing the installation program for your application. Windows does not provide a default installation program, but it does provide an Add/Remove Programs application in Control Panel that helps guide the user through starting the installation, update, or removal process. When the user chooses to install an application, Add/Remove Programs automatically checks the floppy and compact disc read-only memory (CD-ROM) drives for installation programs, searching for filenames such as SETUP.EXE and INSTALL.EXE. If a file is found and the user agrees to finish the installation, Add/Remove Programs starts the program and exits. After that, the started program is responsible for guiding the user through the rest of the installation process.

Designing the User Interface

Your installation program should use the standard Windows graphical user interface. It should present users with options and status. Sections of *The Windows Interface Guidelines for Software Design* describe how to design an application that is consistent with the look and feel of the Windows shell. It will also give you information about easy-to-implement features that will add value to your application and make use of new usability functionality in the shell.

Your installation program should always offer setup options. The following options are recommended.

Typical setup	Installs the application with all of the most common settings and copies the most commonly used files. This should be the default setup option.
Compact setup	Copies the fewest number of files needed to operate your application. This option is useful for laptops and computers on which disk space is at a premium.
Custom setup	Allows the user to determine the details of the installation, such as the directories to receive the files and the application features to enable. This option, which is typically used by the power user, should also include an option to set up components left out during a typical or compact setup.
Silent setup	Runs setup without user interaction. This should just be a command line option so that your installation program can be run within a batch script.

Your installation program should always supply defaults. In particular, it should supply a common response to every option so that all the user has to do is press the ENTER key.

Your installation program should never ask the user to install a disk more than once and should make the computer beep when it is time for the user to insert a new disk.

Your installation program should always include a progress indicator to show users how far along they are in the setup procedure.

Your installation program should always give the user a chance to cancel the setup process before it is finished. Your program should keep a log of files that have been copied and settings that have been made so that it can clean up a canceled installation. If the installation is canceled, your program should remove any registry entries it may have made, remove any shortcuts it may have added to the desktop, and delete any files it may have copied onto the user's hard disk.

Determining the Configuration

Your installation program should determine the hardware and software configuration of the user's computer before copying files and setting the environment. It is important for the installation program to verify that everything needed to successfully run the application is available. For example, if your application depends on specific hardware or software, your installation program should make sure the hardware or software is present. If it is not, the program should notify the user immediately and recommend a course of action.

Your installation program should always tell the user how much disk space is needed. For custom setup, the installation program should adjust the "space needed" figure as the user selects and deselects options. Your installation program should verify that enough disk space is present for the options that the user selects. If there is not enough free space, the program should notify the user but give the user the option to override the warning.

Your installation program should always determine whether any of the files to be installed are already on the hard disk. This is especially important for shared files, such as commonly used dynamic-link libraries (DLL). If the files already exist, your installation program should check the version number to ensure that it is not replacing a file with an older version. In other words, the installation program should always make sure the most recent version of a file is installed on the user's disk.

Copying Files

Your installation program should copy all necessary executable and data files to the appropriate directories. It should never copy these files to the WINDOWS or SYSTEM directories. Instead, it should create a directory in the Program Files directory and copy its files there. If the Program Files directory does not exist on the root of the hard disk, your installation program should create it.

It is recommended that your installation program use a long filename for the directory, such as the application name or another descriptive and unique name. Your program should copy the main executable file for your application and any other executable or data files that the user may want to open directly to the newly created directory. For example, if your application's name is "My Wizzy Application.Exe", your installation program should create the \Program Files\My Wizzy Application directory, and copy My Wizzy Application.Exe to that directory.

If you have any other executable or data files, such as .DLL and .HLP files that are specific to your application, your installation program should create a subdirectory, named System, in your application's directory. It should copy the remaining files (except shared files) to this new directory. For example, if your application has a DLL named MWASUP.DLL, your installation program should create the \Program Files\My Wizzy Application\System directory and copy the DLL there.

If any of your executable or data files are shared, your installation program needs to copy the files to yet another directory, depending on how widely the file is to be shared. A file is *system-wide shared* if many applications from different vendors use it. For example, the VBRUN300.DLL file is a system-wide shared file, because it is used by any application built with Visual Basic. A file is a *shared file* if it is shared by a set of applications from the same vendor. A common example of this would be an office suite that might use the same drawing program for its word processor as it does for its spreadsheet.

Your installation program should copy all system-wide shared files to the Windows SYSTEM directory. If a given file already exists in this directory, the program should overwrite it with your application file *only if* your file is a more recent version. The **GetFileTime**, **GetFileVersionInfo**, and **GetFileInformationByHandle** functions can be used to determine which file is more recent. After copying a DLL file, your installation program should increment the usage counter for the DLL in the registry. For more information about the usage counter, see Adding Entries to the Registry.

Your installation program should copy all shared files to a System directory in the \Program Files\Common Files directory. If the directory does not exist, the installation program should create it. Again, it is recommended that your program use a descriptive and unique name. For example, if there is a shared file named My Wizzy Speller.Exe, your program should create a directory named \Program Files\Common Files\System and copy the file there. The location of the Program Files and Common Files directories is registered (using the macro **REGSTR_PATH_SETUP**) in the **HKEY_LOCAL_MACHINE** root under the S**OFTWARE\Microsoft\Windows\CurrentVersion** key. The value names are ProgramFilesDir and CommonFilesDir.

When your installation program installs applications on computers running Microsoft® Win32s® with Windows version 3.*x*, it needs to be aware that the system does not support long filenames. Your installation program will need to use the short 8.3 filename equivalent for Program Files and Common Files, which is Progra~1 and Common~1, respectively.

Replacing DLLs in Memory

Installation programs often need to replace old .DLL files with new versions. However, what if the DLL is already in memory? It is still possible to replace the DLL. The method you use to replace DLLs in memory depends on the platform.

Windows NT

To replace DLLs already loaded in memory, use the **MoveFileEx** function.

Windows 95

Windows 95 does not allow a .DLL file to be replaced if the DLL is currently loaded into memory. To solve this problem, your installation program must copy the new .DLL files to the user's machine, giving each new .DLL file a temporary name that is different from that of the corresponding old .DLL file. Your installation program must also copy a file called WININIT.INI to the user's machine. The WININIT.INI file is processed by the WININIT.EXE program when the system is restarted, before any DLLs are loaded. The WININIT.INI file specifies the destination path and filename for each new DLL.

The WININIT.INI file contains a [rename] section that specifies the source and destination path and filenames for the new DLLs. The entries in the [rename] section have the following syntax.

```
DestinationFileName=SourceFileName
```

The following syntax is used to delete a file.

```
NUL=SourceFileName
```

The following example shows a [rename] section from a WININIT.INI file.

```
[rename]
C:\WINDOWS\Fonts\arial.ttf=C:\WINDOWS\Fonts\arial.win
C:\WINDOWS\SYSTEM\advapi32.dll=C:\WINDOWS\SYSTEM\advapi32.tmp
```

When the system is restarted, it searches for a WININIT.INI file and, if it finds one, runs WININIT.EXE on the file. After processing the file, WININIT.EXE renames it to WININIT.BAK.

The DestinationFileName and SourceFileName must both be short (8.3) names instead of long filenames because WININIT.EXE is a non-Windows application and runs before the protected mode disk system is loaded. Because long filenames are only visible when the protected mode disk system is loaded, WININIT.EXE won't see them, and therefore, won't process them.

Installing Fonts

By carrying out these steps, you can write a single font installation routine that works for both Windows NT and Windows 95:

1. Determine whether the platform is Windows 95 or Windows NT. This distinction is important because Windows 95 allows a shared network installation where most system files, including fonts, are stored on a centrally managed server. To determine the platform, look in the following registry location for a "SharedDir" value.

   `HKeyLocalMachine\Software\Microsoft\Windows\CurrentVersion\Setup`

   The data value of "SharedDir" is the UNC name of the server and sharepoint of the shared directory. In most cases, a shared directory is marked as read-only by the system administrator, so your installation program should also check to see if it can write to this location. If it cannot, it should let the user install the fonts in a different location, or stop the setup process.

2. Check whether the TrueType® font being installed is already present on the system by using the **EnumFontFamilies** function. If that font is present, the program should check to see if its version is newer by matching the installed font name with the filename on the disk. The font name is stored in the following registry location for both Windows 95 and Windows NT.

   `HKeyLocalMachine\Software\Microsoft\Windows\CurrentVersion\Fonts`

   The subkeys in this registry location contain the full name of the font file as the value key, followed by the filename of the .TTF file as the key data. If the filename in the registry is just a filename with no path information, the font is installed in the \WINDOWS\FONTS directory for Windows 95 or the \WINDOWS\SYSTEM directory for Windows NT. Because TrueType font files do not carry a version resource, your program will need to retrieve the version string from the 'name' table in the .TTF file.

   - Before copying the .TTF file to the appropriate directory, the installation program should check to see if the filename already exists in that directory. If it does, the program should rename your .TTF file to some other name, perhaps by appending a number to the end of the basename.
   - After copying the .TTF file to the user's disk, the installation program should inform the system that it wants the font to be available. The program should pass it the .TTF filename directly by using the **AddFontResource** function. Windows 95 and Windows NT do not require the creation of .FOT files.
   - To make the font installation permanent, the installation program should add the font name and filename to the registry by writing both of the values to the following registry location.

     `HKeyLocalMachine\Software\Microsoft\Windows\CurrentVersion\Fonts`

Removing an Application

Your installation program can direct the Add/Remove Programs application in Control Panel to list your application as an application that can be "automatically removed" by adding the following entries to the registry.
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall*application-name*
DisplayName=*product-name* **UninstallString**=*full-path-to-program command-line-parameters*
Add/Remove Programs displays the product name specified by the **DisplayName** value in its list of applications that can be removed. Windows uses the value specified by the **UninstallString** value to start the uninstall program to carry out the removal of the application. This string needs to completely specify the command-line parameters needed to execute the uninstall program and remove the application. A full path is required. If both the DisplayName and UninstallString values are not complete, Add/Remove Programs will not list the application.
Windows needs to know when the removal of the application is done, so it requires the **UninstallString** value to specify the uninstall program that actually carries out the removal. A batch file or other program that starts the removal program should not be specified.
Your installation program should use casual names, including spaces, for the *application-name* and **DisplayName** value. Casual names help keep the tree comprehensible for users who browse the registry. The registry locations are defined as constants for C programmers in the REGSTR.H header file. Descriptions of the macros follow.

REGSTR_PATH_UNINSTALL	Path to uninstall branch
REGSTR_VAL_UNINSTALLER_DISPLAYNAME	DisplayName
REGSTR_VAL_UNINSTALLER_COMMANDLINE	UninstallString

The uninstall program must display a user interface that informs the user that the removal process is taking place. Your uninstall program should provide a silent option that allows the user to run it remotely. The uninstall program should also display clear and helpful messages for any errors it encounters during the removal of the application. Windows will only detect and report a failure to start the uninstall program.
An uninstall program should complete the following steps:

- Remove all information used by the application from the registry. If decrementing a DLL's usage count results in a usage count of zero, the uninstall program should display a message offering to delete the DLL or save it in case it may be needed later.
- Remove any shortcuts to the application from the desktop.
- Remove all program files related to the application. The uninstall program should not remove files that the user created with the application unless the user agrees to delete them. If the user's files are stored in the application's directory tree, the uninstall program should ask the user if the files should be moved to a new directory.
- Remove empty directories left by the application.

Setting Up the Environment

Your installation program needs to set up the proper environment for your application. The environment consists of application-specific entries in the initialization files, the registry, and the Start menu.

Setting Initialization Files

Windows does not require the AUTOEXEC.BAT and CONFIG.SYS files. Because these files may not be present on the hard disk, you should make sure that your application does not require entries in those files.
Windows does not require you to modify the PATH environment variable. Instead, Windows looks for your .EXE and .DLL files in the application-specific path specified in the registry. Your installation program is responsible for setting the application-specific path when it installs the application.
Windows does not require an application to load device drivers at boot time. This means that your application does not need to specify drivers in the CONFIG.SYS file. Instead, your application can dynamically load the drivers when it starts by using the virtual device loader functions of Windows or the **CreateFile** and **DeviceIoControl** functions.
Your installation program should *not* make entries in the WIN.INI file. It should be adding entries to the registry instead. If you have information that you do not want to put in the registry, your installation program should create a private initialization file and place it in the same directory that contains your application's executable files.

Adding Entries to the Registry

Your installation program should add information about your application to the registry. In particular, it should always add the following entries.

HKEY_LOCAL_MACHINE\SOFTWARE*CompanyName**ProductName**Version*

Stores information pertaining to this particular copy of the application.

HKEY_CURRENT_USER\SOFTWARE

Stores user-specific preferences. This is information that application vendors used to store in the WIN.INI file. For example Microsoft Word might store the fact that a user wants the automatic save feature turned off here.

Your installation program should always add application-specific paths to the registry for your application. If your installation program registers an path, Windows sets the PATH environment to be the registered path when it starts your application. Your program sets the path in the **HKEY_LOCAL_MACHINE** root under the **\SOFTWARE\Microsoft\Windows\CurrentVersion\AppPaths** key (using the **REGSTR_PATH_APPPATHS** macro). Your installation program must create a new key having the same name as your application's executable file. Under this new key, it creates the Path value name and assigns it a path using the same format as expected by the PATH environment variable.

The following example shows application-specific paths for both Windows® Excel, Excel.Exe, and My Wizzy Application.Exe.

```
HKEY_LOCAL_MACHINE
  SOFTWARE\Microsoft\Windows\CurrentVersion\AppPaths
    Excel.Exe
      Default=D:\Program Files\MS Office\Excel\Excel.Exe
      Path= D:\Program Files\MS Office\Excel\Excel.Exe;D:\Program
          Files\Common Files\MS Office;

    My Wizzy App.Exe
         Default=d:\Program Files\My Wizzy Application\My Wizzy
             Application.Exe
         Path= D:\Program Files\My Wizzy Application;D:\Program Files\My
             Wizzy Application\Application Extensions;
```

In the preceding example, the Default value specifies the full path to the corresponding executable file. This value is typically used by Windows in the Start Run command. If the user types the name of your application but Windows fails to find it in the current path, Windows uses this value to locate and start your application.

Your installation program should keep track of shared DLLs. When installing an application that uses shared DLLs, it should increment the usage counter for the DLL in the registry. When removing an application, it should decrement the usage counter. If the result is zero, the user should be given the option of deleting the DLL. The user should be warned that other applications may actually need the DLL and will not work if it is missing. The following example shows the general format for usage counters in the registry.

```
\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\SharedDLLs
    C:\Program Files\Common Files\System\vbrun300.DLL=3
```

Supporting Context Menu Operations

Your installation program can provide support for context menu operations, such as Open, Print, and Print To, by setting appropriate registry entries. The context menu appears when the user clicks mouse button 2 on a document associated with your application.

Enabling Print in the registry gives the shell instructions about what to execute when the user chooses Print from the context menu. Usually an application will display a dialog box that says " Printing page *n* of *N* on LPT*X*."

Enabling Print To in the registry specifies the default action for "drag print." Print To displays the same dialog box as Print when you drag it to a specific printer. The Print To option is not displayed on the context menu, so it does not bring up anything (that is, it cannot be chosen).

The following example shows how to set commands for the context menu for files having the .WRI filename extension.

```
HKEY_CLASSES_ROOT\.wri = wrifile
HKEY_CLASSES_ROOT\wrifile = Write Document
HKEY_CLASSES_ROOT\wrifile\DefaultIcon =
    C:\Progra~1\Access~1\WORDPAD.EXE,2
HKEY_CLASSES_ROOT\wrifile\shell\open\command = WORDPAD.EXE %1
HKEY_CLASSES_ROOT\wrifile\shell\print\command =
    C:\Progra~1\Access~1\WORDPAD.EXE /p "%1"
HKEY_CLASSES_ROOT\wrifile\shell\printto\command =
    C:\Progra~1\Access~1\WORDPAD.EXE /pt "%1" "%2" "%3" "%4"
```

In the preceding commands, the %1 parameter is the filename, %2 is the printer name, %3 is the driver name, and %4 is the port name. In Windows 95, you can ignore the %3 and %4 parameters (the printer name is unique in Windows 95).

Adding the Application to the Start Button

Your installation program can still create a "Program Group" in the Programs folder by using dynamic data exchange (DDE) as used in Windows version 3.1, but this is no longer the preferred method. Instead, your installation program should add an icon for your primary application to the Start menu. The program can, optionally, prompt the user to choose which program icons to place in the menu. However, icons should not be added for every application in your package, and an extensive hierarchy of programs and folders should not be created on the Start menu.

To add an icon to the Start menu, your installation program should create a link to your application's executable file by using the **IShellLink** interface. Place the link in the directory specified by the following registry key:

HKEY_CURRENT_USER\Sofware\Microsoft\Windows
CurrentVersion\Explorer\User shell folders.

Using Filename Extensions

Filename extensions should always describe a file type. Your installation program should not rename old or backup files by giving them filename extensions like .001, .BAK, or .XX1. If the file type does not change, the program should give the file a new name. For example, it can use long filenames to change the old version of a filename, such as SAMPLE.DLL being changed to Copyof SAMPLE.DLL.
The following table lists filename extensions currently used in Windows. You should not use these filename extensions, unless your file fits the given type description.

Extension	Type description
386	Windows virtual device driver
3GR	Screen grabber for Microsoft® MS-DOS® - based applications
GR3	Windows version 3.0 screen grabber
ACM	Audio Compression Manager driver
ADF	Administration configuration files
ANI	Animated mouse cursor
AVI	Video clip
AWD	Fax viewer document
AWP	Fax key viewer
AWS	Fax signature viewer
BAK	Backed-up file
BAT	MS-DOS batch file
BFC	Briefcase
BIN	Binary data file
BMP	Picture (Windows bitmap)
CAB	Windows setup file
CAL	Windows Calendar file
CDA	CD audio track
CFG	Configuration file
CNT	Help contents
COM	MS-DOS -based program
CPD	Fax cover page
CPE	Fax cover page
CPI	International code page
CPL	Control Panel application
CRD	Windows Cardfile document
CSV	Command-separated data file
CUR	Cursor (pointer)
DAT	System data file
DCX	Fax viewer document
DLL	Application extension (dynamic-link library)
DOC	WordPad document
DOS	MS-DOS - based file (also extension for NDIS2 net card and protocol drivers)
DRV	Device driver
EXE	Application

FND	Saved search results
FON	Font file
FOT	Shortcut to font
GR3	Windows version 3.0 screen grabber
GRP	Program group file
HLP	Help file
HT	HyperTerminal file
ICM	Image color matching (ICM) profile
ICO	Icon
IDF	MIDI instrument definition
INF	Setup information
INI	Configuration settings
KBD	Keyboard layout
LGO	Windows logo driver
LIB	Static-link library
LNK	Shortcut
LOG	Log file
MCI	MCI command set
MDB	File viewer extension
MID	MIDI sequence
MIF	MIDI instrument file
MMF	Microsoft Mail message file
MMM	Animation
MPD	Mini-port driver
MSG	Microsoft Exchange mail document
MSN	The Microsoft Network home base
MSP	Windows Paintbrush picture
NLS	Natural language services driver
PAB	Microsoft Exchange personal address book
PCX	Picture (PCX format)
PDR	Port driver
PF	ICM profile
PIF	Shortcut to MS-DOS - based application
PPD	PostScript® printer description file
PRT	Printer formatted file (result of Print to File option)
PST	Microsoft Exchange personal information store
PWL	Password list
QIC	Backup set for Microsoft Backup
REC	Windows Recorder file
REG	Application registration file
RLE	Picture (RLE format)
RMI	MIDI sequence
RTF	Document (rich text format)
SCR	Screen saver
SET	File set for Microsoft Backup

SHB	Shortcut into a document
SHS	Scrap
SPD	PostScript printer description file
SWP	Virtual memory storage
SYS	System file
TIF	Picture (TIFF format)
TMP	Temporary file
TRN	Translation file
TSP	Windows telephony service provider
TTF	TrueType font
TXT	Text document
VBX	Visual Basic control file
VER	Version description file
VXD	Virtual device driver
WAV	Sound wave
WPC	WordPad file converter
WRI	Windows Write document
XAB	Microsoft Mail address book

You should also investigate filename extensions commonly used by popular applications so that you can avoid creating a new extension that might conflict with them, unless you intend to replace or supersede the functionality of those applications.

Register Document Types

Your installation program should register every file type used that is not provided by the system:

- For the files of interest to the user, such as document types, the installation program should register both an icon and a description. It should provide good OLE/shell verbs and also add a "ShellNew" entry so your document type shows up in the "New" menu. This menu is available when the user clicks mouse button 2 on any container or chooses the File menu in a folder window.
- For files that the user would have a good reason to double-click, the installation program should provide the file with a good icon and description and also a registered "open" action so that the user can double-click it.
- For files that are less interesting to the user, such as .INI or configuration files, the installation program should provide the file with a good icon and description. The best way to do this is to consistently use predefined filename extensions, such as .INI, .SYS, and .TXT.
- For files of little interest to the user, the installation program should minimally register a file type so that there is a decent description in "Details" view (and possibly an icon). If the program does not register the type, the file is identified by whatever the filename extension may be. Registering the type ensures that the file is identified by the description and related icon.

Network Issues

Most corporate customers would like to run their applications from a network server. To support running from a server, you need to provide your installation application in both a server and client package. The server package consists of executable files, DLLs, data files, and any files that must be shared across the network. The client package consists of the portions of the application that are user-specific, including registry settings, details about the user's configuration, and information about how to locate the server package.

Generally, you should have two installation programs or modes for installing the packages: an administrative installation program that an administrator runs for preparing the server and a client installation program that runs on each client machine and sets up the connection to the server. The client installation program should also have a batch or silent installation option so that an administrator can deploy your application with automatic software distribution tools. Ideally, the client installation functions are built into the application so that it configures itself when it starts (perhaps by reading options set by the administrative installation program).

Corporate customers typically run Windows from a shared copy on a server. The following directories are stored on the server; your application and client installation program may or may not have write access to these directories.

```
\Windows
    \Command
    \Inf
    \Fonts
    \Help
    \Hyperterm
    \Pif
    \System
        \Color
        \Iosubsys
        \Viewers
        \VMM32
```

You should use the **GetSystemDirectory** function to find the SYSTEM subdirectory. To find the WINDOWS directory, look in the following registry location.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Setup
    SharedDir=
```

Your application should store files that cannot be shared (machine-specific files) in a "machine" directory with write access. The machine directory contains files and settings that are specific to a particular machine. If one user changes settings, anyone else who uses that computer gets those settings. If the machine has user profiles turned on, Windows copies the user-specific settings into and out of the machine directory when the user logs on and off. That way, if a user changes a machine setting (that is, a hardware setting), every user is affected, but if the user changes a user-specific setting, the change affects only that user.

The machine directory should not contain any executable files. You can find the machine directory by calling the **GetWindowsDirectory** function. The following files and directories are stored in the machine directory.

WIN.COM	\Spool
WININIT.EXE	\Desktop
*.INI	\Startmen
*.GRP	\Nethood

Your application and installation program should fully support Universal Naming Convention (UNC) paths. If an application is being installed on a network path, the installation program should store a UNC path in any shortcuts it makes for the Start menu. Your installation program can use the Windows Network (WNet) functions to determine if a path is a network path.

You should consider what configuration settings an administrator might want to set for a user and what restrictions an administrator might want to place on a user (for example, not letting a user access a configuration menu). You should put these settings and restrictions in a System Policy template (.ADM) file.

CD-ROM Considerations

Autorun is a feature that is supported on CD-ROM drives. When the user loads a compact disc (CD) into the drive, the system automatically runs a file on the CD. The file to run must be specified in an AUTORUN.INF file located in the CD's root directory. The following example shows a typical entry in an AUTORUN.INF file.

```
[AutoRun]
OPEN=myprog.exe
```

The autorun feature can be disabled by the device manager or by an entry in the SYSTEM.INI file. Your application must not rely on the autorun feature being available. Also, the autorun feature should not be used to automatically install your application on a user's hard disk without the user being asked first.

If you provide your application on a CD, your installation program should give the user the choice of running the application from the CD or installing it on the hard disk. You should keep the following points in mind when using the autorun feature:

- Even if the user chooses to run your application from the CD, your program will need to copy some files to the hard disk (for example, writable files and files containing the user's preferences).
- If you include a shortcut on the desktop, your application should display a message when the user selects the shortcut and the CD is not loaded.

Checklist for Planning an Installation Program

You should keep the following points in mind when you plan an installation program for your application:

- Do not assume that floppies are on Drive A.
- Use a graphical user interface.
- Always supply defaults.
- Name your installation program SETUP.EXE.
- Tell the user how much space the installation will take and use a progress indicator.
- Make sure to create all directories in the user selected path.
- Store private initialization (.INI) files in the application directory if the application is running locally or in the directory returned by the **GetWindowsDirectory** function if the application is shared.
- Provide an uninstall option.

WINDOWS.H and STRICT Type Checking

The WINDOWS.H include file contains a number of type definitions, macros, and structures that aid in writing source code portable between versions of Microsoft Windows. Some of the WINDOWS.H features are enabled when the STRICT symbol is defined on the command line or makefile. This guide explains how these STRICT features affect the writing of correct code and what the advantages of using them are.

This guide discusses the following major topics:

New Types and Macros

Porting Code from 16-bit Windows to 32-bit Windows introduces some new standard types for Windows programming. Use of the old types, such as **FAR PASCAL** for declaring Windows procedures, may work in existing code but is not guaranteed to work in all future versions of Windows. Therefore, you should convert your code to use the new standards wherever appropriate.

General Data Types

The following table summarizes the new standard types defined in WINDOWS.H. These types are polymorphic (they can contain different kinds of data) and are useful generally throughout applications. There are also other new types, handles, and function pointers that are introduced in following sections.

Typedef	Description
WINAPI	Use in place of **FAR PASCAL** in function declarations. If you are writing a DLL with exported function entry points, you can use this for your own functions.
CALLBACK	Use in place of **FAR PASCAL** in application callback routines such as window procs and dialog procs.
LPCSTR	Same as **LPSTR**, except used for read-only string pointers. Defined as (**const char FAR***).
UINT	Portable unsigned integer type whose size is determined by host environment (32 bits for Windows NT). Synonym for **unsigned int**. Used in place of **WORD** except in the rare cases where a 16-bit unsigned quantity is desired even on 32-bit platforms.
LRESULT	Type used for declaration of Window procedure return value.
WPARAM	Type used for declaration of first general purpose Window procedure parameter.
LPARAM	Type used for declaration of second general purpose Window procedure parameter.
LPVOID	Generic pointer type, equivalent to **(void *)**. Should be used in preference to **LPSTR**.

Utility Macros

WINDOWS.H provides a series of utility macros that are useful for working with the types listed in the previous section. These macros, listed in the following table, help create and extract data from these types. The **FIELDOFFSET** macro is particularly useful when you need to give the numeric offset of a structure member as an argument.

Utility	Description
MAKELPARAM(*low*, *high*)	Combines two 16-bit quantities into an **LPARAM**.
MAKELRESULT(*low*, *high*)	Combines two 16-bit quantities into an **LRESULT**.
MAKELP(*sel*, *off*)	Combines a selector and an offset into a **FAR VOID*** pointer. Useful only for Windows 3.*x*.
SELECTOROF(*lp*)	Extracts the selector part of a far pointer. Returns a **UINT**. Useful only for Windows 3.*x*.
OFFSETOF(*lp*)	Extracts the offset part of a far pointer. Returns a **UINT**. Useful only for Windows 3.*x*.
FIELDOFFSET(*type*, *field*)	Calculates the offset of a member of a data structure. The *type* is the type of structure, and *field* is the name of the structure member or field.

New Handle Types

In addition to the existing Windows handle types such as **HWND**, **HDC**, **HBRUSH**, and so on, WINDOWS.H defines the following new handle types. They are particularly important if STRICT type checking is enabled, but you can use these even if you do not define STRICT.

Handle	Description
HINSTANCE	Instance handle type
HMODULE	Module handle type
HBITMAP	Bitmap handle type
HLOCAL	Local handle type
HGLOBAL	Global handle type
HTASK	Task handle type
HFILE	File handle type
HRSRC	Resource handle type
HGDIOBJ	Generic GDI object handle type (except HMETAFILE)
HMETAFILE	Metafile handle type
HDWP	**DeferWindowPos**() handle
HACCEL	Accelerator table handle
HDRVR	Driver handle (Windows NT only)

Using STRICT to Improve Type Checking

Defining the STRICT symbol enables features that require you to be more careful in declaring and using types. This may sound like a burden, but it is actually an aid to writing more portable code. This extra care will also reduce the time you spend debugging. Enabling STRICT redefines certain data types so that the compiler won't permit assignment from one type to another without an explicit cast. This is especially helpful with Windows code. Errors in passing data types are reported at compile time instead of causing fatal errors at run time.
When STRICT is defined, WINDOWS.H type definitions change as follows:

- Specific handle types are defined so as to be mutually exclusive; for example, you won't be able to pass an **HWND** where an **HDC** type argument is required. Without STRICT, all handles are defined as integers, so the compiler doesn't prevent you from using one type of handle where another type is expected.
- All callback function types (dialog procedures, window procedures, and hook procedures) are defined with full prototypes. This prevents you from declaring callback functions with incorrect parameter lists.
- Parameter and return value types that should use a generic pointer are declared correctly as **LPVOID** instead of as **LPSTR** or other pointer type.
- The **COMSTAT** structure is now declared according to the ANSI standard.

Enabling STRICT Type Checking

To enable STRICT type checking, just define the symbol name "STRICT". You can specify this definition on the command line or in a makefile by giving **/DSTRICT** as a compiler option.
To define STRICT on a file-by-file basis (supported by C but not C++ as explained in the note in this section), insert a **#define** statement before including WINDOWS.H in files where you want to enable STRICT:

```
#define STRICT
#include WINDOWS.H
```

For best results, you should also set the warning level for error messages to at least /W3. This is good policy with Windows applications in any case, because a coding practice that causes a warning (for example, passing the wrong number of parameters) usually causes a fatal error at run time if it is not corrected.

Note If you are writing a C++ application, you don't have the option of applying STRICT to only some of your source files. Because of the way C++ "type safe linking" works, mixing STRICT and non-STRICT source files in your application can cause linking errors.

Making Your Application STRICT Compliant

Some source code that in the past compiled successfully might produce error messages when you enable STRICT type checking. The following sections describe the minimal requirements you need to follow, where applicable, to make sure your code compiles when STRICT is enabled. There are other steps not strictly required but recommended, especially if you want to produce portable code. These are covered in General Requirements

The principal requirement is that you must declare correct handle types and function pointers instead of relying on more general types such as **unsigned int** and **FARPROC**. You cannot use one handle type where another is expected. This requirement also means that you may have to change function declarations and use more type casts.

For best results, the generic HANDLE type should not be used except where necessary. Consult Using Function Pointers

Always declare function pointers with the proper function type (such as **DLGPROC** or **WNDPROC**) rather than **FARPROC**. You'll need to cast function pointers to and from the proper function type when using **MakeProcInstance**, **FreeProcInstance**, and other functions that take or return a **FARPROC**, as shown in the following code:

```
BOOL CALLBACK DlgProc(HWND hwnd, UINT msg, WPARAM wParam,
                                  LPARAM lParam);
DLGPROC lpfnDlg;

lpfnDlg = (DLGPROC)MakeProcInstance(DlgProc, hinst);
...
FreeProcInstance((FARPROC)lpfnDlg);
```

Declaring Functions Within Your Application

Make sure all application functions are declared. Placement of all function declarations in an include file is highly recommended because you can more easily scan through your function declarations and look for parameter and return types that should be changed.

If you use the **/Zg** compiler option to create header files for your functions, remember that you'll get different results depending on whether or not you have enabled STRICT type checking. With STRICT disabled, all handle types generate the same base type: **unsigned short**. With STRICT enabled, they generate base types such as **HWND __near *** or **HDC __near ***. To avoid conflict, you need to either recreate the header file each time you disable or enable STRICT, or else edit the header file to use the types **HWND**, **HDC**, **HANDLE**, and so on, instead of the base types.

If you've copied any function declarations from WINDOWS.H into your source code, they may have changed, and your local declaration may be out of date. Remove your local declaration.

Functions That Require Casts

Some functions have generic return types or parameters. For example, a function like **SendMessage** returns data that may be any number of types, depending on the context. When you see any of these functions in your source code, make sure that you use the correct type cast and that it is as specific as possible.
The following table summarizes these functions:

Function	Comment
LocalLock	Cast result to the proper kind of data pointer.
GlobalLock	Cast result to the proper kind of data pointer.
GetWindowWord	Cast result to appropriate data type.
GetWindowLong	Cast result to appropriate data type
SetWindowWord	Cast argument as it is passed to function.
SetWindowLong	Cast argument as it is passed to function.
SendMessage	Cast result to appropriate data type; cast to **UINT** before casting to a handle type.
DefWindowProc	See comment for **SendMessage**.
SendDlgItemMessage	See comment for **SendMessage**.

When you call **SendMessage**, **DefWindowProc**, or **SendDlgItemMessage**, you should first cast the result to type **UINT**. You need to take similar steps for any function that returns **LRESULT** or **LONG**, where the result contains a handle. This is necessary for writing portable code because the size of a handle is either 16 bits or 32 bits depending on the version of Windows. The **(UINT)** cast ensures proper conversion. The following code shows an example in which **SendMessage** returns a handle to a brush:

```
HBRUSH hbr;

hbr = (HBRUSH)(UINT)SendMessage(hwnd, WM_CTLCOLOR, ..., ...);
```

The CreateWindow Function

The **CreateWindow** and **CreateWindowEx** *hmenu* parameter is sometimes used to pass an integer control ID. In this case, you must cast this to an **HMENU** type:

```
HWND hwnd;
int id;

hwnd = CreateWindow("Button", "Ok", BS_PUSHBUTTON,
        x, y, cx, cy, hwndParent,
        (HMENU)id,      // Cast required here
        hinst,
        NULL);
```

Making Best Use of STRICT Type Checking

To get the most benefit from STRICT type checking, there are other guidelines you should follow in addition to those in Making Your Application STRICT Compliant. Your code will be more portable in future versions of Windows if you make the following changes:

Change	To
HANDLE	A specific handle such as **HINSTANCE**, **HMODULE**, **HGLOBAL**, **HLOCAL**, and so on
WORD	**UINT**, except where you want a 16-bit value even when the platform is 32 bits
WORD	**WPARAM**, where *wParam* is declared
LONG	**LPARAM** or **LRESULT** as appropriate

Any time you need an integer data type, you should declare it as **UINT** except where a 16-bit value is specifically required (as in a structure or parameter). For even if a variable never exceeds the range of a 16-bit integer, it can be more efficiently handled by the processor if it is 32 bits.

The types **WPARAM**, **LPARAM**, **LRESULT**, and **void *** are "polymorphic data types": they hold different kinds of data at different times, even when STRICT type checking is enabled. To get the benefit of type checking, you should cast values of these types as soon as possible. Note that message crackers (as well as the Microsoft Foundation Classes) automatically recast *wParam* and *lParam* for you in a portable way.

Take special care to distinguish **HMODULE** and **HINSTANCE** types. Even with STRICT enabled, they are defined as the same base type. Most kernel module management functions use **HINSTANCE** types, but there are a few functions that return or accept only **HMODULE** types.

Accessing the New COMSTAT Structure

The Windows 3.0 declaration of the **COMSTAT** structure is not compatible with ANSI standards. WINDOWS.H now defines the **COMSTAT** structure to be compatible with ANSI compilers and so that the **/W4** option does not issue warnings.

To support backward compatibility of source code, WINDOWS.H does not use the new structure definition unless the Windows version (as indicated by WINVER) is greater than 3.0 or if STRICT is defined. When you enable STRICT, the presumption is that you are trying to write portable code. Therefore, WINDOWS.H uses the new **COMSTAT** structure for all versions of Windows if STRICT is enabled.

The new structure definition replaces the bit fields by flags accessing bits in a single field, named **status**, as shown below. Each flag turns on a different bit.

Windows 3.0 field name	Flag accessing the status field
fCtsHold	CSTF_CTSHOLD
fDsrHold	CSTF_DSRHOLD
fEof	CSTF_EOF
fRlsdHold	CSTF_RLSDHOLD
fTxim	CSTF_TXIM
fXoffHold	CSTF_XOFFHOLD
fXoffSent	CSTF_XOFFSENT

If you have code that accesses any of these status fields, you need to change your code as appropriate. For example, suppose you have the following code written for Windows 3.0:

```
if (comstat.fEof || fCondition)
      comstat.fCtsHold = TRUE;
      comstat.fTxim = FALSE;
```

This code should be replaced by code that access individual bits of the **status** field by using flags. Note the use of bitwise operators.

```
if ((comstat.status & CSTF_EOF) || fCondition)
      comstat.status |= CSTF_CTSHOLD;
      comstat.status ~= CSTF_TXIM;
```

Interpreting Error Messages Affected by STRICT

Enabling STRICT type checking may affect the kind of error messages you receive. With STRICT enabled, all handle types as well as the types **LRESULT**, **WPARAM**, and **LPARAM** are defined as pointer types. When you incorrectly use these types (for example, passing an **int** where an **HDC** is expected), you will get error messages referring to errors in pointer indirection.

Another effect of STRICT is to require that **FARPROC** function pointers be recast as more specific function pointer types such as **DLGPROC**. However, **MakeProcInstance** and **FreeProcInstance** still work with the **FARPROC** type. If you fail to cast between **FARPROC** and the appropriate function pointer type, the compiler will report an error in function parameter lists.

Note that using **MakeProcInstance** is useful for the sake of portability, if you want to use the same source to compile for Windows 3.*x*. Under Win32, however, **MakeProcInstance** performs no operation, but just returns the function name.

Generic Thunks

Windows NT supports running 16-bit Windows-based applications using a technology referred to as WOW (Windows on Win32). Each 16-bit application is run as a thread of a 32-bit process. Windows 95 also supports running 16-bit Windows-based applications. They run as 16-bit processes.
Neither Windows NT or Windows 95 allow direct mixing of 16-bit code and 32-bit code in the same process. Both platforms support IPC mechanisms, such as DDE, RPC, OLE, named pipes, and WM_COPYDATA, which you can use for communication between 16-bit code and 32-bit code. However, there are occasions when it is necessary to call a function in a Win32-based DLL (including functions in the system DLLs) from a 16-bit application. Generic thunks provide a mechanism for 16-bit applications to call functions in Win32-based DLLs.

About Generic Thunks

The generic thunking mechanism provides functions that allow a 16-bit application to load a Win32-based DLL, get the addresses of its exported functions, call the functions (passing each one up to thirty-two 32-bit arguments), convert 16:16 (WOW) addresses to 0:32 addresses (useful if you need to build up a 32-bit structure that contains pointers and pass a pointer to it), and free the Win32-based DLL. This process is similar to the run-time dynamic linking used between a Win32-based application and a Win32-based DLL.

You may find that other IPC mechanisms are more powerful and portable. Keep this in mind when deciding whether or not to use thunks.

Note Windows 95 provides another thunking mechanism called flat thunks. For more information, see Thunk Compiler.

Process Context

In generic thunking, the Win32-based DLL is run in the context of the process that loaded it, namely the 16-bit Windows-based application.
The GDI, dialog box, message box, and message functions work within 32-bit code loaded in the context of a 16-bit process. However, not all base features are supported in the context of a 16-bit process. In general, 32-bit code loaded by 16-bit processes can use the Win32 heap functions, memory-mapped file functions, file functions, and functions involving the current process and thread. You should avoid using third party Win32-based DLLs, unless you are sure they work safely in a 16 bit-environment.
Windows 95: Under Windows 95, there are a few additional limitations for Win32-based DLLs loaded by a 16-bit application:

- The DLL uses the stack reserved by the 16-bit application, which is much smaller than the 1MB default stack used by a Win32-based application (generally between 5 and 45K).
- The DLL cannot create new threads. Certain Win32 functions, such as those supporting the common dialog boxes or those supporting console applications, create threads on behalf of the calling application. Therefore, these functions cannot be used in a Win32-based DLL loaded by a 16-bit process.

Testing Your Generic Thunks

Debugging generic thunks is difficult, because the debuggers do not debug both the 16-bit and 32-bit sides of the thunk and it is difficult to run both a 16-bit and a 32-bit debugger simultaneously. For this reason, it is best to debug each side separately, then test the thunking portion, as follows.

- Test the Win32-based DLL by calling it from a Win32-based application. Then you will be sure that the functions you will call through the thunk work correctly.
- Test the 16-bit application by having it call a 16-bit DLL. Then you will be sure that you are calling the functions correctly, with the right data.
- Test the thunk by having the 16-bit application call functions in the Win32-based DLL. Start with a simple thunk that does not require parameters. After you know that the basic thunk works, try a thunk that requires parameters. This approach helps you separate problems in calling the thunk from problems in passing parameters.

Be sure to test on both Windows NT and Windows 95. Because of architectural differences between Windows NT and Windows 95, programs using generic thunks may not be portable between the two platforms. In particular,

- A Win32 function that can be called in the context of a 16-bit process under Windows NT is not guaranteed to work under Windows 95, and vice versa. For more information, see Process Context.
- If more than one 16-bit application loads the same Win32-based DLL, you should not rely on them sharing the DLL data . On Windows NT, 16-bit applications that share the same address space use the same DLL instance, so they share its data section, while 16-bit applications that run in separate address spaces use separate copies of the DLL. On Windows 95, 16-bit applications always use separate copies of the DLL.

Using Generic Thunks

One approach for implementing generic thunks is to isolate thunking code into DLLs. The advantage is that you can easily discard the thunking code when it is no longer needed. You can also create separate DLLs for each platform, to isolate platform-specific differences. Another approach for isolating platform differences is to detect the platform at run time and call the appropriate functions for each platform.
The examples in this overview demonstrate the required steps for a 16-bit Windows-based application named APP16 call the MyPrint routine from the Win32-based DLL named DLL32. These examples also give hints for isolating thunking code into a DLL named DLL16.

- Declaring the DLL function
- Loading the Win32-based DLL
- Getting the address of the DLL function
- Calling the DLL function
- Freeing the Win32-based DLL

You can also use generic thunks to call back into the 16-bit side of the thunk from the 32-bit side of the thunk (generic callback). For more information, see **WOWCallback16**.

Note You can also use generic thunks to translate pointers outside of thunks. For more information, see, Translating 16:16 Pointers.

Declaring the DLL Function

You can declare the functions to be called through generic thunks (target functions) with either the standard-call calling convention (Intel only) or the C calling convention. It is important to call the target function using the correct convention.
The following example shows how to define the target function, MyPrint, in DLL32 as a standard-call function, using the WINAPI modifier:

```
void WINAPI MyPrint( LPTSTR lpString, HANDLE hWnd )
{
    ...
}
```

If you are isolating your thunking code into a DLL, create a DLL16 file and define MyPrint in DLL16 as well. Then when you call MyPrint from APP16, you will be calling the version in DLL16 and the version in DLL16 will perform the thunk to DLL32.

Loading the Win32-based DLL

You load the Win32-based DLL, DLL32, from APP16 by using the **LoadLibraryEx32W** function.

```
// Load the Win32-based DLL from the 16-bit code
if( NULL == (ghLib = LoadLibraryEx32W( "dll32.dll", NULL, 0 )) )
{
    MessageBox( NULL, "Cannot load DLL32", "App16", MB_OK );
    return 0;
}
```

If you are isolating your thunking code into DLL16, you can put the call to **LoadLibraryEx32W** in the **LibMain** function of the DLL16 code.
The instance handle is stored in the following global variable:

```
DWORD ghLib;
```

When linking the 16-bit code, you need to indicate that the generic thunking functions will be imported from the system kernel. For example, using Microsoft Visual C++, you would create an IMPORTS section in the module definition (.DEF) file for APP16, as follows.

```
IMPORTS
    kernel.LoadLibraryEx32W
    kernel.FreeLibrary32W
    kernel.GetProcAddress32W
    kernel.GetVDMPointer32W
    kernel.CallProc32W
    kernel.CallProcEx32W
```

Getting the Address of the DLL Function

Before you can call the target function, MyPrint, from the 16-bit code that loaded DLL32, you must get its address using the **GetProcAddress32W** function. For example:

```
// Get the address of the MyPrint routine in the Win32-based DLL
typedef void (FAR PASCAL *MYPROC)(LPSTR);
MYPROC hProc;

if( NULL == (hProc = (MYPROC)GetProcAddress32W( ghLib, "MyPrint" )))
{
    MessageBox( hWnd, "Cannot call DLL function", "App16", MB_OK );
    ...
}
```

If you are isolating your thunking code into DLL16, put the call to **GetProcAddress32W** in the MyPrint function of the DLL16 code.

Calling the DLL Function

Once you have obtained the address of the target DLL function, you can call it from APP16 using the **CallProcEx32W** or **CallProc32W** function. You cannot create a prototype for **CallProc32W** unless you do one of the following:

- Restrict each file so that it only uses calls to functions that contain the same number of parameters.
- Create a prototype CallProc32W_1 for functions that use one parameter, CallProc32W_2 for functions that use two parameters, and so forth. When linking your application or DLL, use forwarders to map calls to these functions to **CallProc32W**.

This is a limitation of the Pascal calling convention. For this reason, you may choose to use the **CallProcEx32W** function, which uses the C calling convention to support a variable number of arguments.
These examples call MyPrint, passing two arguments, a string and a window handle. All parameters must be 32-bit values. Therefore, the string is declared using a FAR pointer, as shown here:

```
char FAR *TestString = "Hello there";
```

You must convert the 16-bit window handle to a 32-bit window handle using the **WOWHandle32** function, as shown here:

```
// Convert the window handle.
DWORD hWnd32;
hWnd32 = WOWHandle32(hWnd, WOW_TYPE_HWND);
```

This first example uses **CallProcEx32W**.

```
// Call the MyPrint routine in the Win32-based DLL
CallProcEx32W( 2 | CPEX_DEST_STDCALL,
                                  2,
                                hProc,
                  (DWORD) TestString,
                               hWnd32);
```

This next example uses **CallProc32W**:

```
// Call the MyPrint routine in the Win32-based DLL
CallProc32W( (DWORD) TestString,
                           hWnd32,
                            hProc,
                               2,
           2 | CPEX_DEST_STDCALL);
```

A mask of 2 (0x10) is given because we want to pass TestString by reference and the handle by value. The system translates the pointer for us.
If you are isolating your thunking code into DLL16, put the call to **CallProc32W** or **CallProcEx32W** in the MyPrint function of the DLL16 code.

Freeing the Win32-based DLL

Be sure to free the Win32-based DLL using the **FreeLibrary32W** function before exiting the code that loaded the DLL (APP16 or DLL16). In APP16, you would make the following call.

```
// Free the Win32-based DLL from the 16-bit Windows-based application.
FreeLibrary32W( ghLib );
```

If you are isolating your thunking code into DLL16, you would put the call in your **WEP** function, if you put the call to the **LoadLibraryEx32W** function in your **LibMain** function.

Translating 16:16 Pointers

You may occasionally need to translate 16:16 pointers to 32-bit pointers. The **GetVDMPointer32W** and **WOWGetVDMPointer** functions accomplish this. Call **GetVDMPointer32W** from 16-bit code and **WOWGetVDMPointer** from 32-bit code to translate 16:16 pointers to 32-bit pointers. In addition, **GlobalFix**, **GlobalUnfix**, **GlobalWire**, and **GlobalUnwire** must be called from 16-bit code. For more information about these functions, see the 16-bit Windows SDK documentation.

Windows 95:

If you use **GetVDMPointer32W** or **WOWGetVDMPointer** and the affected 16:16 pointer represents a movable global memory manager block, it is important that you first call the **GlobalFix** or **GlobalWire** function on the segment portion of the pointer. If you do not, the 16-bit global memory manager can move the block while your process is executing 32-bit code. If this happens, the linear address returned by **GetVDMPointer32W** or **WOWGetVDMPointer** will become invalid. The debugging version of Windows 95 generates warnings about calls to **GetVDMPointer32W** on unfixed segments.

The **WOWGetVDMPointerFix** function is similar to **WOWGetVDMPointer**, but ensures that the memory pointed to will not be moved by the global memory compacter until the **WOWGetVDMPointerUnfix** function has been called. **WOWGetVDMPointerFix** performs an implicit **GlobalFix** operation on the selector, if necessary. If the selector is allocated as a fixed block or if it is not from the global memory manager, no special action is taken. You should use this function instead of calling **GlobalFix** separately, because it is easier to use and is faster than calling both **GlobalFix** and **WOWGetVDMPointer**.

WOWGetVDMPointerUnfix takes a 16:16 address and undoes the effect of **WOWGetVDMPointerFix** on the segment (the offset portion is ignored). This function should be called once the linear address is no longer needed to avoid fragmentation. It is faster than the **GlobalUnfix** function and correctly handles (that is, ignores) selectors that are not from the 16-bit global memory manager.

Windows NT:

While you are executing 32-bit code called through a generic thunk, the 16-bit side of the thunk is completely blocked until you return or yield. Therefore, in most cases, you do not need to fix or wire the memory. Therefore, **WOWGetVDMPointerFix** is identical to **WOWGetVDMPointer** (it does not call **GlobalFix**) and **WOWGetVDMPointerUnfix** has no effect under Windows NT.

If you are targeting only Windows NT and your 32-bit code yields the message system (by calling functions such as **SendMessage**, **BroadcastSystemMessage**, **GetMessage**, **PeekMessage**, **MessageBox**, or **DialogBox**), you should either call **GlobalFix** or **GlobalWire** ahead of time on the 16-bit side or refresh your 32-bit pointers after the yielding function call has completed by calling **WOWGetVDMPointer**.

If you are targeting both Windows NT and Windows 95, there are two approaches that you can use. The easiest approach is to have the 16-bit side of the thunk first fix all of the segments in memory using **GlobalFix** or **GlobalWire**. In the 32-bit code, use **WOWGetVDMPointer**, rather than **WOWGetVDMPointerFix**, because the segments are already fixed in memory. After the 32-bit code returns, call **GlobalFix** or **GlobalUnwire** in the 16-bit code. This guarantees that your 32-bit pointers remain valid, regardless of the platform, even if the 16-bit global memory manager compacts the global heap. The drawback to this approach is that many segments can be left fixed in memory for extended periods of time, potentially causing out-of-memory conditions due to memory fragmentation.

The most efficient approach that still works on both platforms requires that you use **WOWGetVDMPointerFix** and **WOWGetVDMPointerUnfix**. On Windows NT, the 32-bit pointers can still be affected by 16-bit memory movement if the 32-bit code yields the message system. After each call that might yield the messaging system, discard your 32-bit pointers using **WOWGetVDMPointerUnfix**. If you continue to use the pointer, repeat the original call to **WOWGetVDMPointerFix** to refresh the pointer. The drawback to this approach is that it cannot be used if the called function that yields the messaging system uses 32-bit pointers that you provide after yielding the messaging system. If you cannot determine if this will be a problem, the first approach would be a safer choice.

Generic Thunks Reference

The following 16-bit and 32-bit functions are associated with generic thunking.

16-bit Generic Thunk Functions

The following generic thunk functions can be called by 16-bit Windows-based applications and DLLs.

CallProc32W
CallProcEx32W
FreeLibrary32W
GetProcAddress32W
GetVDMPointer32W
LoadLibraryEx32W

CallProc32W

Use the **CallProc32W** function in 16-bit code to call an entry-point function in a 32-bit DLL.
You cannot create a prototype for **CallProc32W** unless you do one of the following:

- Restrict each file so that it only uses calls to functions that contain the same number of parameters.
- Create a prototype CallProc32W_1 for functions that use one parameter, CallProc32W_2 for functions that use two parameters, and so forth. When linking your application or DLL, use forwarders to map calls to these functions to **CallProc32W**.

This is a limitation of the Pascal calling convention. For this reason, you should use the **CallProcEx32W** function, which uses the C calling convention to support a variable number of arguments.

```
DWORD FAR PASCAL CallProc32W(
    DWORD param1,          // parameter for DLL function
    DWORD param2,          // parameter for DLL function
    ... ,                  // up to 32 parameters for DLL function
    LPVOID lpProcAddress32,// the DLL function to be called
    DWORD fAddressConvert, // bit mask
    DWORD nParams          // number of parameters passed
   );
```

Parameters

param1 through *param32*
Parameters for the 32-bit procedure represented by *lpProcAddress32*.

lpProcAddress32
A value corresponding to the procedure to be called, which is returned by the **GetProcAddress32W** function.

fAddressConvert
Bit mask representing which parameters will be treated as 16:16 pointers and translated into flat linear pointers before being passed to the 32-bit procedure. The lowest bit in the mask represents the last parameter specified (*paramN*), the second lowest bit represents the second to the last parameter specified (*paramN-1*), and so on, so that the highest bit in the mask represents *param1*.

nParams
Number of **DWORD** parameters to be passed to the DLL function (*param1* through *paramN*). For functions that take no parameters, this parameter will be zero. You can also specify the calling convention by using the OR operator to combine this value with one of the following constants:

Value	Meaning
CPEX_DEST_STDCALL	The function uses the standard-call calling convention. This is the default.
CPEX_DEST_CDECL	The function uses the C calling convention.

Return Value

Returns the return value from the 32-bit entry-point function represented by *lpProcAddress32*. The return value can also be zero under the following conditions:

- *lpProcAddress32* is zero
- *nParams* is greater than 32
- *lpProcAddress32* returns zero

Remarks

CallProc32W takes at least three parameters: *lpProcAddress32*, *fAddressConvert*, and *nParams*. In addition, it can take a maximum of 32 optional parameters. These parameters must be **DWORD** types and must match the type that the 32-bit thunk DLL is expecting. If the appropriate bit is set in the *fAddressConvert* mask, the parameter will be translated from a 16:16 pointer to a 32-bit flat linear pointer.

CallProc32W and **CallProcEx32W** do not automatically fix global memory handles that are translated to 0:32 pointers. Therefore, you must call the **GlobalFix** or **GlobalWire** function on the handle first and **GlobalUnfix** and **GlobalUnwire** afterward.

Windows 95: Global compaction can move memory blocks at any time while the current thread is executing 32-bit code. Because of this, not fixing segments before calling the target function works in Windows NT, but may cause race conditions in Windows 95.

Note You should be careful when using this function, because there is no compiler check made on the number and type of parameters, no conversion of types (all parameters are passed as type DWORD and are passed directly to the function without conversion). No checks of the 16:16 address are made (limit checks, NULL checks, correct ring level, and so on).

See Also

CallProcEx32W, **GetProcAddress32W**

CallProcEx32W

Use the **CallProcEx32W** function in 16-bit code to call an entry-point function in a 32-bit DLL. **CallProcEx32W** is similar to **CallProc32W**, but it uses the C calling convention to allow a variable number of arguments.

```
  DWORD FAR CallProcEx32W(
....DWORD nParams,          // number of parameters passed
    DWORD fAddressConvert,  // bit mask
    DWORD lpProcAddress,    // the DLL function to be called
    DWORD param1            // parameter for DLL function
    ...                     // up to 32 parameters for DLL function
   );
```

Parameters

nParams

Number of **DWORD** parameters to be passed to the DLL function (*param1* through *paramN*). For functions that take no parameters, this parameter will be zero. You can also specify the calling convention by using the OR operator to combine this value with one of the following constants:

Value	Meaning
CPEX_DEST_STDCALL	The function uses the standard-call calling convention. This is the default.
CPEX_DEST_CDECL	The function uses the C calling convention.

fAddressConvert

Bit mask representing which parameters will be treated as 16:16 pointers and translated into flat linear pointers before being passed to the 32-bit procedure. The lowest bit in the mask represents the first parameter specified (*param1*), the second lowest bit represents the second to the last parameter specified (*param2*), and so on, so that the highest bit in the mask represents *paramN*.

lpProcAddress

A value corresponding to the procedure to be called, which is returned by the **GetProcAddress32W** function.

param1 through *param32*

Parameters for the 32-bit procedure represented by *lpProcAddress*

Return Value

Returns the return value from the 32-bit entry-point function represented by *lpProcAddress*. The return value can also be zero under the following conditions:

- *lpProcAddress* is zero
- *nParams* is greater than 32
- *lpProcAddress* returns zero

Remarks

CallProc32W and **CallProcEx32W** do not automatically fix global memory handles that are translated to 0:32 pointers. Therefore, you must call the **GlobalFix** or **GlobalWire** function on the handle first and **GlobalUnfix** and **GlobalUnwire** afterward.

Windows 95: Global compaction can move memory blocks at any time while the current thread is executing 32-bit code. Because of this, not fixing segments before calling the target function works in Windows NT, but may cause race conditions in Windows 95.

See Also

CallProc32W, **GetProcAddress32W**

FreeLibrary32W

Use the **FreeLibrary32W** function in 16-bit code to free a 32-bit thunk DLL that it had previously loaded with the **LoadLibraryEx32W** function.

```
BOOL FAR PASCAL FreeLibrary32W(
    DWORD hInst                 // handle to loaded library module
);
```

Parameters

hInst

Identifies the loaded library module. Returned by **LoadLibraryEx32W**.

Return Value

Returns TRUE if successful or FALSE otherwise.

Remarks

The system does not do any cleanup of 32-bit thunk DLLs when the 16-bit application exits. The 16-bit application or DLL must free the 32-bit thunk DLL when it is finished using it.

Windows 95: This function causes Windows 95 to release the Win16Mutex. If this function is called from a 16-bit DLL, it can be reentered as soon as the 32-bit DLL entry-point function is called. This happens when another 32-bit process thunks to the 16-bit DLL or when another 16-bit DLL calls this 16-bit DLL. However, if this function is called in a 16-bit application and the 32-bit DLL entry-point function does not yield, the function is not reentered.

See Also

LoadLibraryEx32W

GetProcAddress32W

Use the **GetProcAddress32W** in 16-bit code retrieve a value that represents a function in a 32-bit DLL.

```
DWORD FAR PASCAL GetProcAddress32W(
    DWORD hInst,            // handle to loaded library module
    LPCSTR lpszProc         // name of function
  );
```

Parameters

hModule

Identifies the loaded library module that contains the function. The **LoadLibraryEx32W** function returns this handle.

lpszProc

Points to a null-terminated string containing the function name.

Return Value

Returns a 32-bit value if successful. This value must be passed as a parameter to the **CallProc32W** or **CallProcEx32W** function rather than being used directly.

Remarks

Windows 95: This function causes Windows 95 to release the Win16Mutex. If this function is called from a 16-bit DLL, it can be reentered as soon as the 32-bit DLL entry-point function is called. This happens when another 32-bit process thunks to the 16-bit DLL or when another 16-bit DLL calls this 16-bit DLL. However, if this function is called in a 16-bit application and the 32-bit DLL entry-point function does not yield, the function is not reentered.

See Also

CallProc32W, **CallProcEx32W**, **LoadLibraryEx32W**

GetVDMPointer32W

Use **GetVDMPointer32W** in 16-bit code to translate a 16:16 pointer into a 32-bit pointer.

```
DWORD FAR PASCAL GetVDMPointer32W(
    LPVOID lpAddress,       // 16:16 address
    UINT fMode              // mode flag
);
```

Parameters

lpAddress

Valid protected or real mode 16:16 address.

fMode

One of the following flags:

1	The address is interpreted as a protected-mode address.
0	The address is interpreted as a real-mode address.

Return Value

Returns a 32-bit linear address if successful or NULL otherwise.

Remarks

On non-x86 platforms, real mode address 0:0 may not point to linear 0 in memory, so always use **GetVDMPointer32W** to avoid making assumptions about memory layout.

The memory manager moves segments in linear memory, but keeps the selectors the same. However, if you get the linear address of a block, it may not be valid if the 16-bit global memory manager moves the block the selector points to.

Windows 95: You should assume that global compaction can occur any time that a generic thunk is entered, a Win32 function is called, or the current application yields.

LoadLibraryEx32W

Use the **LoadLibraryEx32W** function in 16-bit code to load a 32-bit DLL.

```
DWORD FAR PASCAL LoadLibraryEx32W(
    LPCSTR lpszLibFile,     // points to name of executable module
    DWORD hFile,            // reserved, must be NULL
    DWORD dwFlags           // entry-point execution flag
  );
```

Parameters

lpszLibFile

Points to a null-terminated string that names a Win32-based DLL module.

hFile

This parameter is reserved for future use. It must be NULL.

dwFlags

Specifies the action to take when loading the module. This parameter can be one of the following values:

DONT_RESOLVE_DLL_REFERENCES
LOAD_LIBRARY_AS_DATAFILE
LOAD_WITH_ALTERED_SEARCH_PATH

For more information on these values, see **LoadLibraryEx**.

Return Value

If the function succeeds, the return value is a 32-bit handle to a DLL module. If the function fails, the return value is NULL.

Remarks

After calling this function, the 16-bit thunk DLL can call the **GetProcAddress32W** function to get the address of the 32-bit entrypoint function(s) and then call the thunk(s) by using the **CallProc32W** or **CallProcEx32W** function. **Windows 95:** This function causes Windows 95 to release the Win16Mutex. If this function is called from a 16-bit DLL, it can be reentered as soon as the 32-bit DLL entry-point function is called. This happens when another 32-bit process thunks to the 16-bit DLL or when another 16-bit DLL calls this 16-bit DLL. However, if this function is called in a 16-bit application and the 32-bit DLL entry-point function does not yield, the function is not reentered.

See Also

CallProc32W, **CallProcEx32W**, **GetProcAddress32W**

32-bit Generic Thunk Functions

The following generic thunk functions can be called by Win32-based applications.

WOWCallback16
WOWCallback16Ex
WOWGetVDMPointer
WOWGetVDMPointerFix
WOWGetVDMPointerUnfix
WOWGlobalAlloc16
WOWGlobalAllocLock16
WOWGlobalFree16
WOWGlobalLock16
WOWGlobalLockSize16
WOWGlobalUnlock16
WOWGlobalUnlockFree16
WOWHandle16
WOWHandle32

WOWCallback16

Use the **WOWCallback16** function in 32-bit code called from 16-bit code (through generic thunks) to call back to the 16-bit side (generic callback).

```
DWORD WINAPI WOWCallback16(
    DWORD vpfn16,          // pointer to callback function
    DWORD dwParam          // parameter for callback function
   );
```

Parameters

vpfn16

A 16:16 pointer to 16-bit callback routine, passed from the 16-bit side.

dwParam

Parameter for the 16-bit callback routine. If this value is a pointer, it can be used as either a 16:16 or 0:32 pointer, as long as both sides agree on the semantics.

Return Value

The return value comes from the callback routine. If the callback routine returns a **WORD** type instead of a **DWORD** type, the upper 16 bits of the return value are undefined and should be ignored by using **LOWORD**(*dwRetCode*). If the callback routine has no return value, the entire return value of this function is undefined.

Remarks

The 16-bit function to be called must be declared with one of the following types.

```
DWORD FAR PASCAL CallbackRoutine(DWORD dwParam);

DWORD FAR PASCAL CallbackRoutine(VOID FAR *vp);
```

The type used is determined by whether the parameter is a pointer.
If you are passing a pointer, you will need to get the pointer by using either the **WOWGlobalAlloc16** or **WOWGlobalAllocLock16** function.

See Also

LOWORD, **WOWGlobalAlloc16**, **WOWGlobalAllocLock16**

WOWCallback16Ex

Use the **WOWCallback16Ex** function in 32-bit code called from 16-bit code (through generic thunks) to call back to the 16-bit side (generic callback).

```
BOOL WINAPI WOWCallback16Ex(
    DWORD vpfn16,          // pointer to callback function
    DWORD dwFlags,         // flags
    DWORD cbArgs,          // byte count of pArgs arguments
    PVOID pArgs,           // arguments for callback function
    PDWORD pdwRetCode      // callback function return code
   );
```

Parameters

vpfn16

A 16:16 pointer to 16-bit callback routine, passed from the 16-bit side.

dwFlags

One of the following flags:

WCB16_CDECL	Calls a _cdecl callback routine.
WCB16_PASCAL	Calls a _pascal callback routine (default).

cbArgs

Count of bytes in arguments (used to properly clean the 16-bit stack).

pArgs

Arguments for the callback routine.

pdwRetCode

Receives the return code from the callback routine.

Return Value

If *cbArgs* is larger than the WCB16_MAX_ARGS bytes that the system supports, the return value is FALSE and the **GetLastError** function returns the ERROR_INVALID_PARAMETER value. Otherwise, the return value is TRUE and the **DWORD** pointed to by *pdwRetCode* contains the return code from the callback routine. If the callback routine returns a **WORD** type instead of a **DWORD** type, the upper 16 bits of the return code are undefined and should be ignored by using **LOWORD**(*dwRetCode*).

Remarks

WOWCallback16Ex allows any combination of arguments up to WCB16_MAX_CBARGS bytes total to be passed to the 16-bit callback routine. Regardless of the value of *cbArgs*, WCB16_MAX_CBARGS bytes will always be copied from *pArgs* to the 16-bit stack. If *pArgs* is less than WCB16_MAX_CBARGS bytes from the end of a page and the next page is inaccessible, **WOWCallback16Ex** will incur an access violation.

The arguments pointed to by *pArgs* must be in the correct order for the callback routine's calling convention. For example, to call a Pascal routine, place the arguments in the *pArgs* array in reverse order, with the least significant word first for **DWORD** types and offset first for **FAR** pointers.

When you call a _cdecl routine, place the arguments in the *pArgs* array in the order listed in the function prototype with the least significant word first for **DWORD** types and offset first for **FAR** pointers.

See Also

LOWORD

WOWGetVDMPointer

The **WOWGetVDMPointer** function converts a 16:16 address to the equivalent linear address.

```
LPVOID WINAPI WOWGetVDMPointer(
    DWORD vp,                   // 16:16 address
      DWORD dwBytes,            // size of vp block
    BOOL fProtectedMode         // protected mode flag
   );
```

Parameters

vp
Valid 16:16 address.

dwBytes
Size of the block pointed to by *vp*.

fProtectedMode
One of the following flags:

1	The upper 16 bits are treated as a selector in the local descriptor table (16-bit protected mode pointer).
0	The upper 16 bits are treated as a real-mode segment value (real-mode 16:16 pointer).

Return Value

Returns a 32-bit address if successful. If the function is invalid, the return value is NULL.

Remarks

Windows NT: Limit checking is performed only in the checked (debugging) build of WOW32.DLL, which will cause NULL to be returned when the limit is exceeded by the supplied offset.
Windows 95: This function should never be used on a 16-bit global memory handle selector that has not been previously fixed in memory by using the **GlobalFix** or **GlobalWire** function. You should assume that global compaction can occur at any time the Win16Mutex is not owned by the current thread.

WOWGetVDMPointerFix

The **WOWGetVDMPointerFix** function converts a 16:16 address to the equivalent linear address.

Windows 95: This function calls the **GlobalFix** function before returning the linear address so that the 16-bit memory block will not be moved around in the 16-bit global heap.

Windows NT: This function behaves like the **WOWGetVDMPointer** function. The memory is not fixed.

```
LPVOID WINAPI WOWGetVDMPointerFix(
    DWORD vp,                   // 16:16 address
      DWORD dwBytes,            // size of vp block
    BOOL fProtectedMode         // protected mode flag
   );
```

Parameters

vp

Valid 16:16 address.

dwBytes

Size of the block pointed to by *vp*.

fProtectedMode

One of the following flags:

1	The upper 16 bits are treated as a selector in the local descriptor table (16-bit protected mode pointer).
0	The upper 16 bits are treated as a real-mode segment value (real-mode 16:16 pointer).

Return Value

Returns a 32-bit address if successful. If the selector is invalid, the return value is NULL.

See Also

WOWGetVDMPointer

WOWGetVDMPointerUnfix

Windows 95: The **WOWGetVDMPointerUnfix** function uses the **GlobalUnfix** function to unfix a pointer retrieved by the **WOWGetVDMPointerFix** function.
Windows NT: This function has no effect.

```
VOID WINAPI WOWGetVDMPointerUnfix(
    LPCSTR vp                   // 32-bit linear address
);
```

Parameters

vp
Address retrieved by the **WOWGetVDMPointerFix** function.

Return Value

This function does not return a value.

See Also

WOWGetVDMPointerFix

WOWGlobalAlloc16

The **WOWGlobalAlloc16** function allocates the specified number of bytes from the 16-bit global heap.

```
WORD WINAPI WOWGlobalAlloc16(
    WORD wFlags,            // object allocation attributes
    DWORD cb                // number of bytes to allocate
   );
```

Parameters

wFlags

Specifies how to allocate memory. This parameter can be a combination of the following values: GHND, GMEM_DDESHARE, GMEM_DISCARDABLE, GMEM_FIXED, GMEM_LOWER, GMEM_MOVEABLE, GMEM_NOCOMPACT, GMEM_NODISCARD, GMEM_NOT_BANKED, GMEM_NOTIFY, GMEM_SHARE, GMEM_ZEROINIT, and GPTR.

cb

Specifies the number of bytes to allocate.

Return Value

Returns the handle of the newly allocated memory object if successful. Otherwise, returns NULL.

See Also

WOWGlobalLock16

WOWGlobalAllocLock16

The **WOWGlobalAllocLock16** function combines the functionality of the **WOWGlobalAlloc16** and **WOWGlobalLock16** functions.

```
DWORD WINAPI WOWGlobalAllocLock16(
    WORD wFlags,          // object allocation flags
    DWORD cb,             // number of bytes to allocate
    LPWORD phMem          // handle to global memory object
  );
```

Parameters

wFlags

Specifies how to allocate memory. This parameter can be a combination of the following values: GHND, GMEM_DDESHARE, GMEM_DISCARDABLE, GMEM_FIXED, GMEM_LOWER, GMEM_MOVEABLE, GMEM_NOCOMPACT, GMEM_NODISCARD, GMEM_NOT_BANKED, GMEM_NOTIFY, GMEM_SHARE, GMEM_ZEROINIT, and GPTR.

cb

Specifies the number of bytes to allocate.

phMem

Handle to the object in the 16-bit global heap. This value is returned by **WOWGlobalAllocLock16**.

Return Value

Returns a pointer to the first byte of the memory block if successful. Otherwise, returns NULL.

Remarks

The pointer returned is a 16:16 pointer that cannot be dereferenced directly in 32-bit code. Instead, call the **WOWGetVDMPointerFix** function.

See Also

WOWGetVDMPointerFix, **WOWGlobalAlloc16**, **WOWGlobalLock16**

WOWGlobalFree16

The **WOWGlobalFree16** function frees the specified global memory object.

```
WORD WINAPI WOWGlobalFree16(
    WORD hMem                   // handle to the global memory object
);
```

Parameters

hMem

Handle to the object in the 16-bit global heap. This value must have been obtained from the **WOWGlobalAlloc16** or **WOWGlobalAllocLock16** function.

Return Value

Returns NULL if successful.

See Also

WOWGlobalAlloc16, **WOWGlobalAllocLock16**

WOWGlobalLock16

The **WOWGlobalLock16** function locks a global memory object and returns a pointer to the first byte of the object's memory block..

```
DWORD WINAPI WOWGlobalLock16(
    WORD hMem                   // handle to the global memory object
);
```

Parameters

hMem

Handle to the object in the 16-bit global heap. This value must have been obtained from the **WOWGlobalAlloc16** or **WOWGlobalAllocLock16** function.

Return Value

Returns a pointer to the first byte of the object's memory block if successful. Otherwise, returns NULL.

Remarks

The pointer returned is a 16:16 pointer that cannot be dereferenced directly in 32-bit code. Instead, call the **WOWGetVDMPointerFix** function.

See Also

WOWGetVDMPointerFix, **WOWGlobalAlloc16**, **WOWGlobalAllocLock16**

WOWGlobalLockSize16

The **WOWGlobalLockSize16** function combines the functionality of the **WOWGlobalLock16** and 16-bit **GlobalSize** functions.

```
DWORD WINAPI WOWGlobalLockSize16(
    WORD hMem,              // handle to the global memory object
    PDWORD pcb              // gets size of the global memory object
  );
```

Parameters

hMem

Handle to the object in the 16-bit global heap. This value must have been obtained from the **WOWGlobalAlloc16** or **WOWGlobalAllocLock16** function.

pcb

Receives the size of the object in the 16-bit global heap.

Return Value

Returns a pointer to the first byte of the memory block if successful. Otherwise, returns NULL.

See Also

WOWGlobalAlloc16, **WOWGlobalAllocLock16**, **WOWGlobalLock16**

WOWGlobalUnlock16

The **WOWGlobalUnlock16** function unlocks a global memory object.

```
BOOL WINAPI WOWGlobalUnlock16(
  WORD hMem                 // handle to the global memory object
);
```

Parameters

hMem

Handle to the object in the 16-bit global heap. This value must have been obtained from the **WOWGlobalAlloc16** or **WOWGlobalAllocLock16** function.

Return Value

Returns zero if the object's lock count was decremented (decreased by one) to zero. Otherwise, returns a nonzero value.

WOWGlobalUnlockFree16

The **WOWGlobalUnlockFree16** function combines the functionality of the **WOWGlobalUnlock16** and **WOWGlobalFree16** functions.

```
WORD WINAPI WOWGlobalUnlockFree16(
    DWORD vpMem             // address of memory block
);
```

Parameters

vpMem

Address of memory block returned by the **WOWGlobalLock16** or **WOWGlobalAllocLock16** function.

Return Value

Returns NULL if successful.

See Also

WOWGlobalAllocLock16, **WOWGlobalFree16**, **WOWGlobalLock16**, **WOWGlobalUnlock16**

WOWHandle16

The **WOWHandle16** function is used to map a 32-bit handle to a 16-bit handle. Because the relationship between a Win16 handle and a Win32 handle may change in the future, use this function to convert handles instead of any knowledge of the relationship between them.

```
WORD WINAPI WOWHandle16(
    HANDLE Handle,                  // 32-bit handle
    WOW_HANDLE_TYPE Type            // handle type
   );
```

Parameters

Handle

The 32-bit handle to be mapped.

Type

Indicates the type of handle being translated. The accepted values are of the form WOW_TYPE_*handle* where *handle* can be **HWND**, **HMENU**, **HDWP**, **HDROP**, **HDC**, **HFONT**, **HMETAFILE**, **HRGN**, **HBITMAP**, **HBRUSH**, **HPALETTE**, **HPEN**, **HACCEL**, and **HTASK**. For example, use WOW_TYPE_HWND to convert an **HWND**.

Return Value

A 16-bit handle.

Remarks

You can also use supplied macros to map handles. For example, to map a 32-bit **HWND** to a 16-bit **HWND**, you would use the **HWND_16** macro.

```
(hWnd16 = HWND_16(hWnd32))
```

WOWHandle16 with WOW_TYPE_HTASK (and **HTASK_16**) takes a thread identifier and converts it to a handle to a task if possible.

WOWHandle32

The **WOWHandle32** function is used to map a 16-bit handle to a 32-bit handle. Because the relationship between a Win16 handle and a Win32 handle may change in the future, use this function to convert handles instead of any knowledge of the relationship between them.

```
HANDLE WINAPI WOWHandle32(
    WORD Handle,                    // 16-bit handle
    WOW_HANDLE_TYPE Type            // handle type
  );
```

Parameters

Handle

The 16-bit handle to be mapped.

Type

Indicates the type of handle being translated. The accepted values are of the form WOW_TYPE_*handle* where *handle* can be **HWND**, **HMENU**, **HDWP**, **HDROP**, **HDC**, **HFONT**, **HMETAFILE**, **HRGN**, **HBITMAP**, **HBRUSH**, **HPALETTE**, **HPEN**, **HACCEL**, **HTASK**, and **FULLHWND**. For example, use WOW_TYPE_HWND to convert an **HWND**.

Return Value

A 32-bit handle.

Remarks

You can also use supplied macros to map handles. For example, to map a 16-bit **HWND** to a 32-bit **HWND**, you would use the **HWND_32** macro.

```
(hWnd32 = HWND_32(hWnd16))
```

The type WOW_TYPE_FULLHWND is a window handle that the system passes to a Win32-based application. The type WOW_TYPE_HWND has a different value, but it is recognized by the system and may be passed as a parameter to Win32 functions. If you intend to store the window handle and use it in comparisions with 32-bit window handles received from Win32 functions, use WOW_TYPE_FULLHWND when calling **WOWHandle32**. Do not make assumptions about the relationship between the 16-bit window handle, the 32-bit window handle, and the full window handle. This relationship has changed in the past (for performance reasons), and it may change again in the future.

When you pass **WOWHandle32** type WOW_TYPE_HTASK with a 16-bit task handle, it returns a 32-bit thread identifier. You may compare this value with other thread identifiers, such as those returned by the **GetWindowThreadProcessId** function.